UNIVERSITA` DEGLI STUDI DI PISA
FACOLTA` DI INGEGNERIA
ISTITUTO DI ELETTRONICA E TELECOMUNICAZIONI

PAOLO CORSINI

MICROPROCESSORE mE86 E CALCOLATORE EB86

MANUALE D'USO

SERVIZIO EDITORIALE UNIVERSITARIO DI PISA

MICROPROCESSORE mE86
E CALCOLATORE EB86

MANUALE D'USO

UNIVERSITA' DEGLI STUDI DI PISA
FACOLTA' DI INGEGNERIA
ISTITUTO DI ELETTRONICA E TELECOMUNICAZIONI

PAOLO CORSINI

MICROPROCESSORE mE86 E CALCOLATORE EB86

MANUALE D'USO

SEU SERVIZIO EDITORIALE UNIVERSITARIO DI PISA

© SEU - Vicolo della Croce Rossa 5 - 56126 Pisa - tel. 40015

Febbraio 1989

Prefazione

In questa dispensa viene presentato il microprocessore didattico mE86 ed il calcolatore su singola scheda EB86 che su tale microprocessore è basato. Viene altresì presentato un ambiente di sviluppo software su host con servizi di editazione, assemblaggio e manutenzione di file: un monitor sul calcolatore EB86 permette il collaudo di programmi utente.

Il microprocessore mE86 ed il calcolatore EB86 sono stati simulati e la loro simulazione, così come l'ambiente di sviluppo, è disponibile sotto UNIX V e sotto DOS 3.30.

Il materiale di questa dispensa costituisce parte del corso di Reti Logiche della Facoltà di Ingegneria ed è stato messo a punto con la collaborazione dell'ing. M. Malcontenti.

ARCHITETTURA DEL MICROPROCESSORE mE86

1. PRESENTAZIONE DEL MICROPROCESSORE mE86

Il microprocessore mE86 è un microprocessore didattico che può essere visto come una versione semplificata del ben noto INTEL iAPX 8086. Il set di istruzioni del microprocessore mE86 è un sottoinsieme di quello dello iAPX 8086, ma non vi è compatibilità a livello di linguaggio macchina in quanto i codici operativi delle istruzioni omologhe hanno una diversa codifica: il linguaggio macchina del microprocessore mE86 è tenuto riservato, per cui le varie istruzioni e le varie modalità di indirizzamento saranno illustrate in un linguaggio mnemonico che sarà introdotto in modo informale e graduale contestualmente al suo utilizzo.

Il microprocessore è in grado di elaborare operandi ad 8 bit (*byte*) ed a 16 bit (*word*) ed in alcuni casi a 32 bit (*doubleword*). Nell'eseguire le istruzioni aritmetiche, il microprocessore interpreta gli operandi come numeri naturali (*unsigned*) oppure come interi con segno (*integer*): la base di lavoro è la *base* 2 e la rappresentazione interna per gli integer è il *complemento a due.*

2. VISIBILITA` A LIVELLO DI LINGUAGGIO MNEMONICO DI UN CALCOLATORE BASATO SUL MICROPROCESSORE mE86

Un calcolatore basato sul microprocessore mE86 appare ad un programmatore che operi a livello di linguaggio mnemonico come illustrato in Fig. 1, dove sono evidenziati oltre ai *registri* del microprocessore anche

lo *spazio di memoria* e lo *spazio di I/O* a cui esso è in grado di accedere.

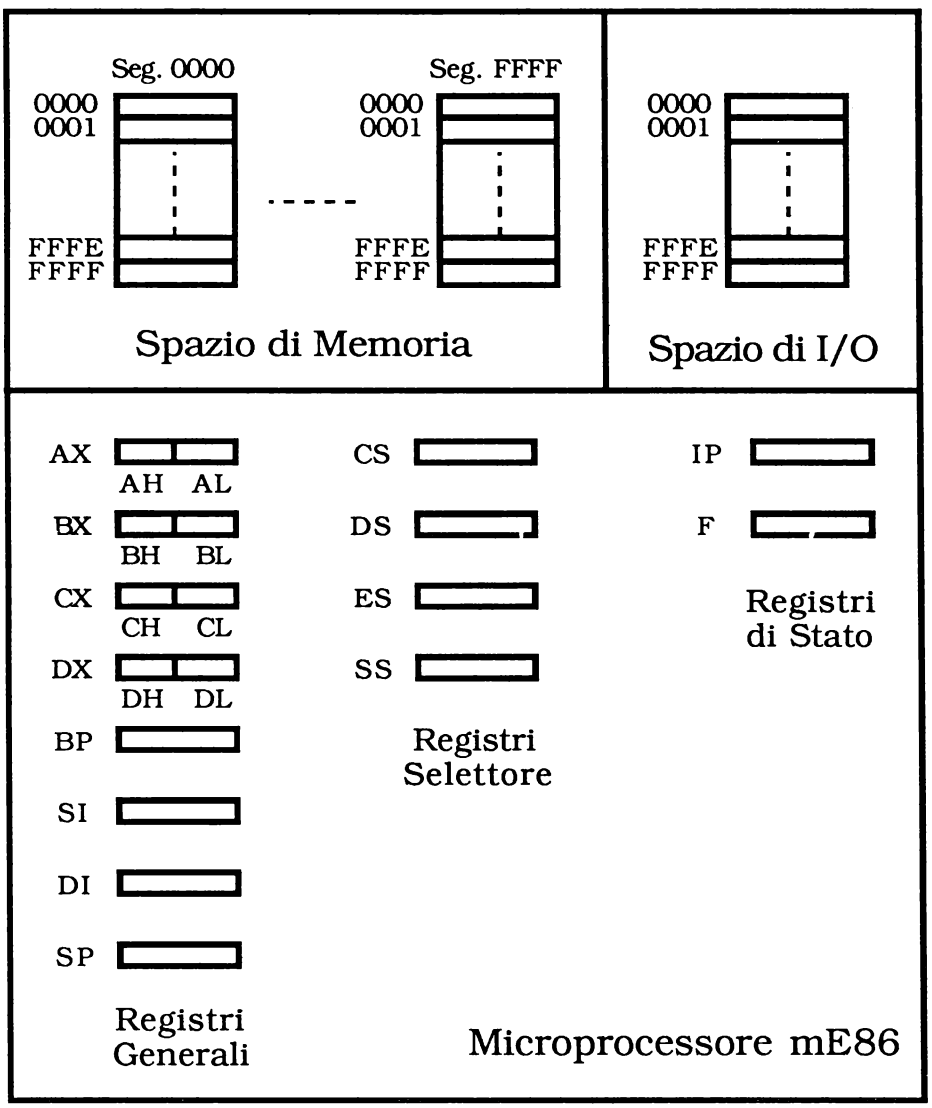


Fig. 1 - Modello di un calcolatore basato sul microprocessore mE86.

LO SPAZIO DI MEMORIA

Lo spazio di memoria (o *memoria logica*) è un insieme di *segmenti* di lunghezza variabile, ciascuno composto da una sequenza lineare e contigua di *locazioni*: ogni locazione ha una capacità pari ad un byte ed è individuabile mediante un *indirizzo logico* costituito da due componenti, il *selettore* e l'*offset*. Il selettore (a 16 bit) individua uno specifico segmento nello spazio di memoria, mentre l'offset (anche esso a 16 bit) individua la locazione all'interno del segmento. Tutte le istruzioni che riferiscono un operando in memoria o che controllano il flusso del programma devono quindi specificare un indirizzo completo nelle due componenti: selettore ed offset.

Un operando, quando risiede in un segmento della memoria, può occupare una o due locazioni; un operando a 8 bit occupa ovviamente una sola locazione mentre un operando 16 bit occupa due locazioni consecutive: in questo caso la prima delle due locazioni, cioè quella di offset più piccolo, contiene la parte meno significativa dell'operando.

La memoria logica è molto grande: i segmenti sono infatti 65536 e ciascuno ha fino a 64 K locazioni per una capacità totale di 2^{32} byte.

In realtà la memoria effettivamente disponibile ha una capacità molto minore in quanto un circuito interno al microprocessore (detto Memory Management Unit MMU) mappa, in modo trasparente al programmatore, la memoria logica in una *memoria fisica lineare* avente una capacità pari ad 1 Mbyte: più precisamente il segmento di selettore *xyzt* viene mappato nella memoria fisica a partire dalla locazione di indirizzo *xyzt·16* (vedi Fig. 2).

In generale un indirizzo logico a due componenti è tradotto in un *indirizzo fisico* a 20 bit in accordo alla relazione:

indirizzo_fisico = selettore · 16 + offset .

Ne risulta che segmenti teoricamente distinti in memoria logica possono essere **sovrapposti** in memoria fisica e di ciò il programmatore deve essere cosciente.

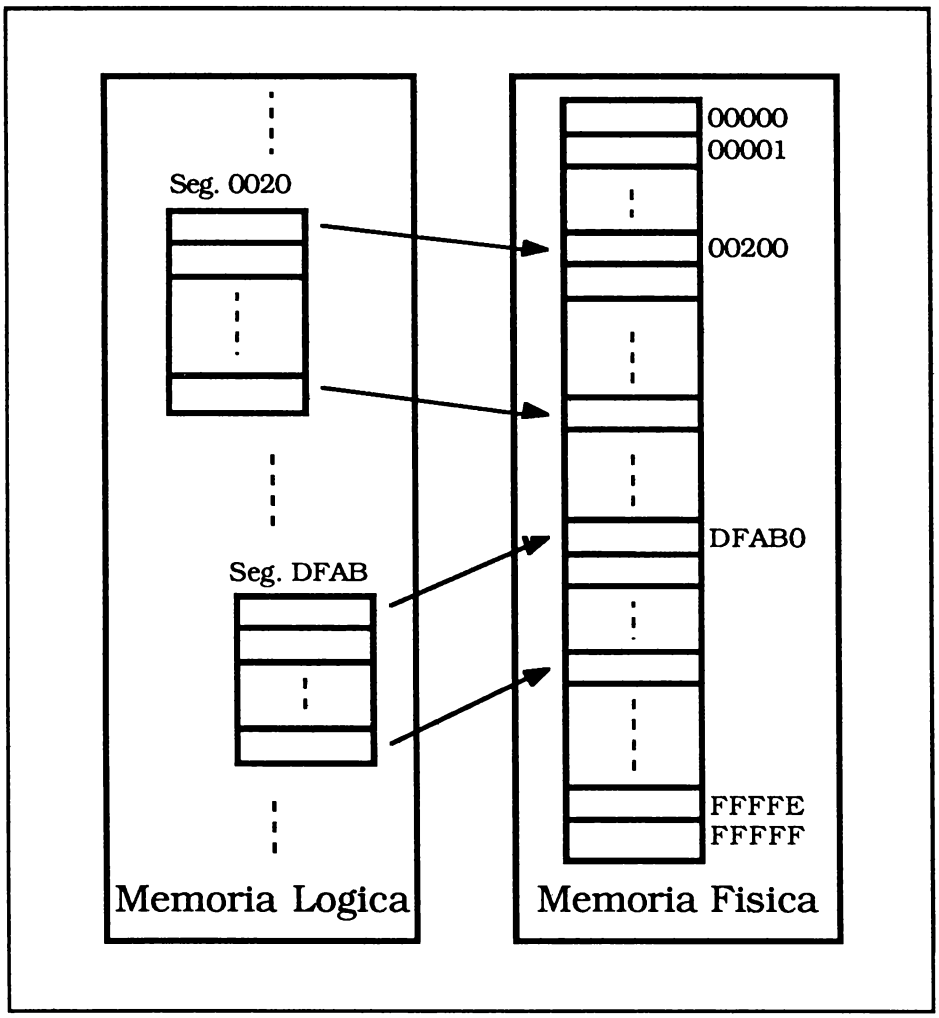


Fig. 2 - Mappaggio della memoria logica nella memoria fisica.

La memoria fisica è implementata in (piccola) parte con integrati di tipo EPROM ed in (gran) parte con integrati di tipo RAM.

LO SPAZIO DI I/O

Lo spazio di I/O è lineare e costituito da 64 K locazioni o *porte* ognuna delle quali ha una capacità di 8 bit ed è indirizzabile attraverso un offset a 16 bit. Le istruzioni che riferiscono un operando nello spazio di I/O (istruzioni di ingresso/uscita) non devono quindi specificare un indirizzo a due componenti, bensì il solo offset.

Quando un operando risiede nello spazio di I/O, può occupare una o due porte; un operando a 8 bit occupa ovviamente una sola porta mentre un operando 16 bit occupa due porte consecutive: in questo caso la prima delle due porte, cioè quella di offset più piccolo, contiene la parte meno significativa dell'operando.

Lo spazio di I/O contiene le interfacce integrate attraverso cui un sistema basato sul microprocessore mE86 colloquia con il mondo esterno: una semplice interfaccia sarà presentata in seguito.

I REGISTRI DEL MICROPROCESSORE

Il microprocessore mE86 possiede tre insiemi di registri: i registri generali, i registri selettore ed i registri di stato.

Registri generali

I *registri generali*, che vengono utilizzati per memorizzare operandi oppure per contenere offset, sono 8 e sono denominati AX, BX, CX, DX, BP, SI, DI e SP. Ciascun registro ha una capacità di 16 bit; i primi 4 registri (AX, DX, CX e BX) sono però divisibili in una parte alta (H) e in una parte bassa (L), e possono quindi essere utilizzati come 8 registri da un byte.

I registri generali vengono utilizzati da alcune istruzioni in modo specifico, per cui risultano spesso dedicati a particolari funzioni. Più precisamente si ha:

- il registro AX (oppure AH od AL) è utilizzato da alcune istruzioni come *accumulatore* (e con questo nome viene riferito);
- i registri DX e AX sono utilizzati dalle istruzioni di moltiplicazione, di divisione e di ingresso/uscita; l'insieme dei due registri DX ed AX visti come contenere un unico operando a 32 bit è detto *accumulatore esteso* DX_AX;
- il registro CX è utilizzato da alcune istruzioni di controllo del flusso del programma e dalle istruzioni di traslazione e rotazione;
- i registri BX, BP, SI e DI sono utilizzati sia come *registri puntatore* che come *registri indice* (vedi le modalità di indirizzamento);
- il registro SP è il puntatore al *top* della *pila*.

Registri selettore

I contenuti di quattro registri sono interpretati dal microprocessore come selettori di altrettanti segmenti: tali registri, noti come CS, DS, ES ed SS sono detti pertanto *registri selettore*. I quattrc segmenti individuati dai registri selettore sono detti a loro volta *segmenti correnti* in quanto direttamente accessibili dal microprocessore: le istruzioni operative e molte tra quelle che controllano il flusso del programma specificano infatti la sola componente offset di un indirizzo logico, mentre come componente selettore viene automaticamente considerato il contenuto di uno dei registri selettore.

Il registro CS (*Code Selector Register*) contiene il selettore del *segmento codice corrente* e viene utilizzato durante la fase di chiamata delle istruzioni. La maggior parte delle istruzioni che controllano il flusso del programma specificano solo un nuovo contenuto per IP e la loro esecuzione non comporta la modifica del contenuto di CS: tali istruzioni mantengono quindi il flusso del programma all'interno del segmento codice corrente. Altre istruzioni specificano anche il selettore di un nuovo segmento codice cosicché la loro esecuzione comporta la modifica del contenuto di CS e quindi il

trasferimento del flusso del programma all'interno ad un altro segmento.

Il registro SS (*Stack Selector Register*) contiene il selettore del *segmento stack* nel quale risiede la *pila* (o *stack*), e viene utilizzato principalmente dalle istruzioni che vi immettono (*push*) e vi prelevano (*pop*) dati secondo la disciplina LIFO (Last In - First Out).

I registri DS (*Data Selector Register*) ed ES (*Extra Selector Register*) contengono i selettori di due *segmenti dati* e vengono utilizzati per il prelievo o la memorizzazione di operandi durante la fase di esecuzione delle istruzioni. I contenuti dei registri DS ed ES possono venir modificati dal programma in qualsiasi momento, con la conseguente possibilità di cambiare uno o entrambi i segmenti contenenti i dati da elaborare.

I valori iniziali dei registri SS, DS ed ES vengono predisposti dal programma che li utilizza. Il valore iniziale del registro CS è invece caricato dalla istruzione che passa il controllo al programma stesso.

Registri di stato e controllo

Questo gruppo di registri comprende lo *Instruction Pointer* IP ed il *Registro dei Flag* F.

Il registro IP contiene l'offset della locazione del segmento codice corrente, a partire dalla quale sarà prelevata la prossima istruzione da eseguire. La coppia di valori contenuta nei registri CS, IP rappresenta quindi l'indirizzo completo della nuova istruzione.
Il registro F (vedi Fig. 3) contiene 5 *flag.*

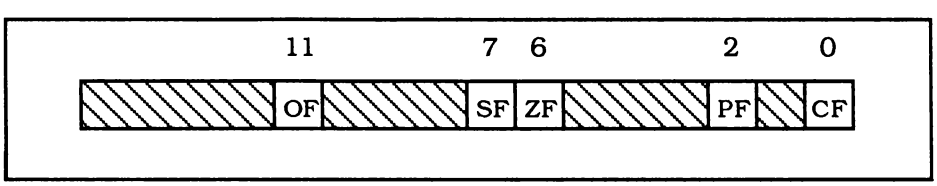


Fig. 3 - Il registro dei Flag F.

La loro funzione è la seguente:

Carry Flag CF: se settato indica che durante l'esecuzione dell'ultima istruzione si è generato un riporto (*carry*) in operazioni di somma o si è richiesto un prestito (*borrow*) in operazioni di sottrazione tra numeri unsigned.

Parity Flag PF: se settato indica che l'ultima istruzione eseguita ha prodotto un risultato con un numero pari di bit 1 nei suoi 8 bit meno significativi.

Zero Flag ZF: se settato indica che l'ultima istruzione eseguita ha prodotto un risultato con tutti i bit uguali a 0.

Sign Flag SF: se settato indica che l'ultima istruzione eseguita ha prodotto un risultato con il bit più significativo uguale ad 1..

Overflow Flag OF: se settato indica che durante l'esecuzione dell'ultima istruzione si è avuto un *traboccamento* in operazioni aritmetiche tra numeri integer.

CONDIZIONI AL RESET INIZIALE

Al reset il microprocessore si porta in uno stato iniziale in cui i registri selettore ed i registri di stato e controllo hanno i seguenti contenuti:

Contenuto del registro F	0000H
Contenuto del registro IP	0000H
Contenuto del registro CS	FFFFH
Contenuto del registro DS	0000H
Contenuto del registro ES	0000H
Contenuto del registro SS	0000H

Ne risulta che l'istruzione che viene eseguita per prima è quella allocata a partire dalla prima locazione del segmento di selettore FFFFH.

3. MODALITÀ DI INDIRIZZAMENTO

Ogni istruzione che viene eseguita dal processore

specifica, oltre al tipo di operazione da eseguire (*codice operativo*), anche le modalità per individuare gli operandi da elaborare (*modalità di indirizzamento*).

Per descrivere le modalità di indirizzamento possibili con il microprocessore mE86, viene introdotto in modo informale un **linguaggio mnemonico** che è in corrispondenza biunivoca con il linguaggio macchina, ma che ha il pregio di una maggiore leggibilità. In tale linguaggio mnemonico uno *statement* del tipo

```
ELAB destination
```

indica una istruzione con codice operativo `ELAB` che agisce su un solo operando elaborandolo e modificandolo; le informazioni `destination` specificano invece le modalità di indirizzamento relative all'operando stesso.

Di seguito vengono illustrate le varie modalità di indirizzamento permesse dal microprocessore mE86.

Modalità registro

Questa modalità di indirizzamento prevede che l'operando sia contenuto in uno degli 8 registri generali da 16 bit (AX, BX, CX, DX, SI, DI, SP, BP), in uno degli 8 registri da 8 bit (AH, BH, CH, DH, AL, BL, CL, DL) ovvero (per alcune istruzioni speciali) in uno dei registri selettore (CS, DS, ES, SS). I seguenti esempi chiariscono come la modalità registro viene indicata a livello di linguaggio mnemonico:

```
ELAB AX
```
Legenda: l'operando ha 16 bit ed è contenuto nel registro AX.

```
ELAB CL
```
Legenda: l'operando ha 8 bit ed è contenuto nel registro CL.

Indirizzamento diretto

Questa modalità di indirizzamento si utilizza quando l'operando è in un segmento di memoria. Essa prevede

che l'istruzione, oltre a specificare quale registro selettore individua il segmento, contenga pure un unsigned a 16 bit detto *displacement* che viene interpretato come l'offset della prima delle locazioni in cui l'operando è allocato. Un caso limite di indirizzamento diretto prevede che l'istruzione contenga anche il selettore del segmento. I seguenti esempi chiariscono come questa modalità di indirizzamento venga indicata a livello di linguaggio mnemonico:

```
ELAB BYTE OPERAND DS:2000H
```
Legenda: l'operando ha 8 bit ed è contenuto nella locazione di offset 2000H all'interno del segmento specificato dal registro DS.

```
ELAB WORD OPERAND ES:2001H
```
Legenda: l'operando ha 16 bit ed è contenuto in due locazioni consecutive del segmento specificato dal registro ES. La prima locazione ha offset 2001H e contiene la parte meno significativa dell'operando

Si noti come le parole chiave `BYTE OPERAND` e `WORD OPERAND` siano indispensabili per specificare il numero di bit dell'operando.

L'indirizzamento diretto è possibile anche quando l'operando risiede nello spazio di I/O, purché ci si limiti a voler accedere alle prime 256 porte. A livello di linguaggio mnemonico, tale situazione può essere descritta come:

```
ELAB BYTE OPERAND IO:70H
```
Legenda: l'operando ha 8 bit ed è contenuto nella porta di offset 70H.

Indirizzamento indiretto con registro puntatore:

Anche questa modalità di indirizzamento si utilizza quando l'operando è in un segmento di memoria. Essa prevede che l'istruzione specifichi il registro selettore che individua il segmento ed un *registro puntatore:* il contenuto di tale registro viene interpretato come l'offset della prima delle locazioni in cui l'operando è allocato: i possibili registri puntatori sono BX, BP, SI e

DI. I seguenti esempi chiariscono come questa modalità di indirizzamento venga indicata a livello di linguaggio mnemonico:

```
ELAB BYTE OPERAND DS:[DI]
```

Legenda: l'operando ha 8 bit ed è contenuto in una locazione del segmento specificato dal registro DS. L'offset di questa locazione è a sua volta contenuto nel registro puntatore DI.

```
ELAB WORD OPERAND ES:[BX]
```

Legenda: l'operando ha 16 bit ed è contenuto in due locazioni consecutive del segmento specificato dal registro ES. L'offset della prima di queste locazioni è a sua volta contenuto nel registro puntatore BX.

L'indirizzamento indiretto con registro puntatore può essere utilizzato per accedere in sequenza ad un vettore di operandi che occupano locazioni adiacenti in memoria: semplici operazioni di aggiornamento del contenuto del registro fanno passare da un elemento al successivo (o al precedente).

Per motivi di speditezza l'indicazione del registro selettore può essere omessa in alcuni casi, valendo la seguente regola di default: qualora non sia specificato il contrario, viene assunto come registro selettore il registro **DS** a meno che il registro puntatore sia **BP**, nel qual caso viene assunto come registro selettore **SS**.

L'indirizzamento indiretto è possibile anche quando l'operando risiede nello spazio di I/O, purché ci si limiti ad usare come registro puntatore il registro **DX**. A livello di linguaggio mnemonico, tale situazione può essere descritta come:

```
ELAB BYTE OPERAND IO:[DX]
```

Legenda: l'operando ha 8 bit ed è contenuto nella porta il cui offset è a sua volta contenuto nel registro puntatore DX.

Indirizzamento con registro indice

Questa modalità di indirizzamento, usabile anche essa quando l'operando è in memoria, rappresenta una

generalizzazione delle due precedenti modalità. Essa prevede che l'istruzione specifichi il registro selettore che individua il segmento, un *displacement* ed un *registro indice:* il displacement sommato al contenuto del registro (visto anche esso come un unsigned) viene interpretato come l'offset della prima delle locazioni in cui l'operando è allocato. I possibili registri indice sono BX, BP, SI e DI. I seguenti esempi chiariscono come questa modalità di indirizzamento venga indicata a livello di linguaggio mnemonico:

```
ELAB BYTE OPERAND DS:2000H[DI]
```

Legenda: l'operando ha 8 bit ed è contenuto in una locazione del segmento specificato dal registro DS. L'offset di questa locazione è ottenibile sommando 2000H al contenuto del registro indice DI.

```
ELAB WORD OPERAND ES:3A2BH[SI]
```

Legenda: l'operando ha 16 bit ed è contenuto in due locazioni consecutive del segmento specificato dal registro ES. L'offset della prima di queste locazioni è ottenibile sommando 3A2BH al contenuto del registro indice SI.

Anche l'indirizzamento con registro indice viene utilizzato per accedere agli elementi di un vettore: il displacement rappresenta l'indirizzo di partenza del vettore (indirizzo del primo elemento), mentre il valore contenuto nel registro indice seleziona uno specifico elemento: semplici operazioni di aggiornamento del contenuto del registro fanno passare, anche in tal caso, da un elemento del vettore al successivo (o al precedente).

Indirizzamento immediato

Molte istruzioni elaborano due operandi e sostituiscono uno di essi con il risultato della elaborazione: l'operando che alla fine viene modificato è detto ***operando destinatario***, l'altro operando (che rimane inalterato) è detto ***operando sorgente***. A livello di linguaggio mnemonico una istruzione a due operandi ha il formato

```
ELAB destination, source
```

Le informazioni `destination` specificano le modalità di indirizzamento relative all'operando destinatario, mentre le informazioni `source` specificano le modalità di indirizzamento relative all'operando sorgente.

Per l'operando sorgente, oltre alle modalità di indirizzamento viste in precedenza, è possibile una ulteriore modalità, detta indirizzamento immediato: essa prevede che l'operando sia contenuto nell'istruzione stessa. I seguenti esempi chiariscono come questa modalità di indirizzamento venga indicata a livello di linguaggio mnemonico

```
ELAB AL, 20H
```

Legenda: gli operandi sono ad 8 bit; l'operando destinatario è individuato mediante la modalità registro ed è contenuto in AL; l'operando sorgente è individuato mediante la modalità di indirizzamento immediato e vale 20H.

```
ELAB WORD OPERAND ES:3A2BH[BX], 32H
```

Legenda: gli operandi sono a 16 bit; l'operando destinatario è individuato mediante la modalità con registro indice ; l'operando sorgente è individuato mediante la modalità di indirizzamento immediato e vale 0032H.

Si noti come nel secondo esempio le parole chiave `WORD OPERAND` siano indispensabili a chiarire che l'istruzione lavora con operandi a 16 bit.

L'indirizzamento immediato è comunemente usato quando l'operando sorgente è una costante da combinare con il contenuto dell'operando destinatario.

IL SET DELLE ISTRUZIONI DEL MICROPROCESSORE mE86

1. CLASSI DI ISTRUZIONI DEL MICROPROCESSORE mE86

Il microprocessore è dotato di un set di istruzioni molto ampio e significativo: tale set è ripartibile nelle tipiche 6 classi:

- Istruzioni per il trasferimento di dati.
- Istruzioni aritmetiche.
- Istruzioni di traslazione/rotazione.
- Istruzioni logiche.
- Istruzioni per il controllo del flusso del programma.
- Istruzioni per il controllo del processore.

Istruzioni per il trasferimento di dati

Le istruzione di trasferimento sostituiscono l'operando destinatario con l'operando sorgente. Sono considerate istruzioni di trasferimento anche le istruzioni per manipolare lo stack. Globalmente le istruzioni per il trasferimento dei dati sono:

MOV	Movimento (ricopiamento)
PUSH	Immissione di una parola nello stack
POP	Estrazione di una parola dallo stack
XCHG	Scambio di posto
IN	Ingresso dati
OUT	Uscita dati

Istruzioni aritmetiche

Queste istruzioni implementano le comuni operazioni aritmetiche interpretando gli operandi come unsigned e/o integer. Le istruzioni di aritmetiche sono

globalmente le seguenti:

ADD	Somma
ADC	Somma con riporto
INC	Incremento di uno
SUB	Sottrazione
SBB	Sottrazione con prestito
DEC	Decremento di uno
NEG	Calcolo dell'opposto
CMP	Confronto
MUL	Moltiplicazione fra unsigned
IMUL	Moltiplicazione fra integer
DIV	Divisione fra unsigned
IDIV	Divisione fra integer
CBW	Conversione di byte in parole
CWD	Conversione di parole in doppie parole

Istruzioni di traslazione/rotazione

Le istruzioni di traslazione o shift effettuano lo spostamento di ogni bit dell'operando nella posizione adiacente a sinistra o a destra. Le istruzioni di rotazione effettuano traslazioni circolari, nel senso che il bit meno significativo e quello più significativo sono supposti adiacenti; le rotazioni possono anche includere il flag CF. Possono essere fatti sino a 255 passi di traslazione/rotazione. Le traslazioni logiche (aritmetiche) possono essere usate per moltiplicare e dividere unsigned (integer) per potenze di 2. Le istruzioni di traslazione/rotazione sono le seguenti:

SHL	Traslazione logica a sinistra
SAL	Traslazione aritmetica a sinistra
SHR	Traslazione logica a destra
SAR	Traslazione aritmetica a destra
ROL	Rotazione a sinistra
ROR	Rotazione a destra
RCL	Rotazione a sinistra tramite il carry
RCR	Rotazione a destra tramite il carry

Istruzioni logiche

Queste istruzioni implementano le più comuni operazione logiche. L' istruzione TEST equivale ad una istruzione AND con la differenza che lascia inalterato l'operando destinatario ed agisce solo sul registro dei flag F. Le istruzioni logiche sono globalmente le seguenti:

NOT	Not bit a bit
AND	And bit a bit
OR	Or bit a bit
XOR	Xor bit a bit
TEST	Test

Istruzioni per il controllo del flusso del programma

Questa classe comprende le istruzioni di salto condizionato, di salto incondizionato, di chiamata a sottoprogramma e di ritorno da sottoprogramma. Negli ultimi tre casi l'istruzione che controlla il flusso del programma e quella che viene messa in esecuzione possono appartenere anche a differenti segmenti codice. Fra le istruzioni di salto condizionato ve ne sono alcune dette **di loop**, che sono particolarmente adatte a programmare situazioni che prevedono cicli. Globalmente le istruzioni che controllano il flusso del programma sono le seguenti:

JMP	Salto incondizionato
CALL	Chiamata di sottoprogramma
RET	Ritorno da sottoprogramma
Jcondition	Salto sotto condizione
LOOPcondition	Ciclo sotto condizione

Istruzioni per il controllo del processore

Questa classe comprende due sole istruzioni:

NOP	Nessuna operazione
HLT	Halt

2. IL SET COMPLETO DELLE ISTRUZIONI DEL MICROPROCESSORE mE86

Le istruzioni del microprocessore mE86 elaborano uno o due operandi, per cui il loro formato a livello di linguaggio mnemonico è normalmente del tipo:

```
ELAB destination
ELAB destination, source
```

In alcune istruzioni a due operandi, quali la PUSH, la POP, la MUL, la DIV etc., le informazioni destination sono implicite nel codice operativo, per cui il formato è in tal caso:

```
ELAB source
```

A livello di linguaggio macchina, ogni istruzione è codificabile con un numero di byte da un minimo di 1 ad un massimo di 6: ad esempio è di 1 byte la codifica dell'istruzione NOP, mentre è di 6 byte la codifica dell'istruzione MOV ES:203AH, 5F21.

Nelle pagine seguenti il set di le istruzioni del microprocessore mE86 viene illustrato in dettaglio.

MOVE

FORMATO: **MOV** *destination, source*

AZIONE: Sostituisce l'operando destinatario con una copia dell'operando sorgente.

FLAG: Non viene modificato alcun flag.

Operandi	Esempi
Memoria, Registro	MOV DS:2000H, DX
Registro, Memoria	MOV CL, ES:1024H
Registro, Registro	MOV AX, DX
Memoria, Immediato	MOV BYTE OPERAND DS:[DI], 5BH
Registro, Immediato	MOV AX, 54A3H
Memoria, Registro_Selettore	MOV DS:[DI], CS
Registro_Selettore, Memoria	MOV SS, ES:1024H
Registro_Selettore, Registro	MOV DS, AX
Registro, Registro_Selettore	MOV BX, ES

PUSH

FORMATO: **PUSH** *source*

AZIONE: Immette nella pila corrente una copia dell'operando sorgente che deve essere a 16 bit. Più in dettaglio: decrementa il registro SP di due e di poi memorizza una copia dell'operando sorgente nel segmento stack (individuato dal contenuto del registro SS) a partire dalla locazione il cui offset è specificato dal contenuto del registro SP.

FLAG: Non viene modificato alcun flag.

Operandi	Esempi
Memoria	PUSH DS:2000H
Registro	PUSH BX
Registro_Selettore	PUSH DS

POP

FORMATO: **POP** *destination*

AZIONE: Rimuove dalla pila corrente una parola e la sostituisce all'operando destinatario. Più in dettaglio: sostituisce all'operando destinatario una copia del contenuto di due locazioni del segmento stack (individuato dal contenuto del registro SS), l'offset della prima delle quali è specificato dal contenuto del registro SP; di poi il contenuto di SP è incrementato di due, col che la parola copiata è rimossa dalla pila.

FLAG: Non viene modificato alcun flag.

Operandi	Esempi
Memoria	POP DS:2000H
Registro	POP BX
Registro_Selettore	POP DS

EXCHANGE

FORMATO: **XCHG** *destination, source*

AZIONE: Sostituisce all'operando destinatario una copia dell'operando sorgente e viceversa.

FLAG: Non viene modificato alcun flag.

Operandi	Esempi
Memoria, Registro	XCHG DS:2000H, DX
Registro, Registro	XCHG AX, DX

INPUT

FORMATO: **IN** AX, IO:*displacement*
IN AL, IO:*displacement*
IN AX, IO:[DX]
IN AL, IO:[DX]

AZIONE: Sostituisce il contenuto di AX (di AL) con il contenuto di due porte consecutive (di una porta). L'offset della prima (ed eventualmente unica porta) è rappresentato dal displacement presente nell'istruzione (primi due formati) o dal contenuto di DX (ultimi due formati); i primi due formati possono essere utilizzati solo per individuare porte con offset inferiore a 256.

FLAG: Non viene modificato alcun flag.

OUTPUT

FORMATO: **OUT** IO:*displacement*, AX
OUT IO:*displacement*, AL
OUT IO:[DX], AX
OUT IO:[DX], AL

AZIONE: Copia il contenuto di AX (di AL) in due porte consecutive (in una porta). L'offset della prima (ed eventualmente unica porta) è rappresentato dal displacement presente nella istruzione (primi due formati) o dal contenuto di DX (ultimi due formati); i primi due formati possono essere utilizzati solo per individuare porte con offset inferiore a 256.

FLAG: Non viene modificato alcun flag.

ADD

FORMATO: **ADD** *destination, source*

AZIONE: Modifica l'operando destinatario sommandovi l'operando sorgente; entrambi gli operandi possono essere interpretati sia come unsigned che come integer. Il flag CF (il flag OF) viene settato se interpretando gli operandi come unsigned (come integer) c'è stato un traboccamento.

FLAG: Tutti i flag vengono modificati.

Operandi	Esempi
Memoria, Registro	ADD DS:2000H, DX
Registro, Memoria	ADD CL, ES:1024H
Registro, Registro	ADD AX, DX
Memoria, Immediato	ADD BYTE OPERAND DS:[DI], 5BH
Registro, Immediato	ADD AX, 54A3H

ADD WITH CARRY

FORMATO: **ADC** *destination, source*

AZIONE: Modifica l'operando destinatario sommandovi prima l'operando sorgente e di poi 1 se il flag CF è settato; entrambi gli operandi possono essere interpretati sia come unsigned che come integer. Il flag CF (il flag OF) viene settato se interpretando gli operandi come unsigned (come integer) c'è stato un traboccamento. Poiché l' istruzione ADC gestisce un riporto che può essere stato generato da una precedente istruzione, essa può essere usata per scrivere routine che sommano numeri più lunghi di 16 bit.

FLAG: Tutti i flag vengono modificati.

Operandi	Esempi
Memoria, Registro	ADC DS:2000H, DX
Registro, Memoria	ADC CL, ES:1024H
Registro, Registro	ADC AX, DX
Memoria, Immediato	ADC BYTE OPERAND DS:[DI], 5BH
Registro, Immediato	ADC AX, 54A3H

INCREMENT

FORMATO: **INC** *destination*

AZIONE: Modifica l'operando destinatario sommandovi 1; l'operando può essere interpretato sia come unsigned che come integer. Il flag OF viene settato se interpretando l'operando come integer c'è stato un traboccamento; il flag CF non viene modificato.

FLAG: Vengono modificati i flag OF, PF, SF e ZF.

Operandi	Esempi
Memoria	INC BYTE OPERAND DS:[SI]
Registro	INC CX

SUBTRACT

FORMATO: **SUB** *destination, source*

AZIONE: Modifica l'operando destinatario sottraendovi l'operando sorgente; entrambi gli operandi possono essere interpretati sia come unsigned che come integer. Il flag CF viene settato se interpretando gli operandi come unsigned è stato richiesto un prestito; il flag OF è settato se interpretando gli operandi come integer c'è stato un trabocca-mento.

FLAG: Tutti i flag vengono modificati.

Operandi	Esempi
Memoria, Registro	SUB DS:2000H, DX
Registro, Memoria	SUB CL, ES:1024H
Registro, Registro	SUB AX, DX
Memoria, Immediato	SUB BYTE OPERAND DS:[DI], 5BH
Registro, Immediato	SUB AX, 54A3H

SUBTRACT WITH BORROW

FORMATO: **SBB** *destination, source*

AZIONE: Modifica l'operando destinatario sottraendovi prima l'operando sorgente e di poi 1 se il flag CF è settato; entrambi gli operandi possono essere interpretati sia come unsigned che come integer. Il flag CF (il flag OF) viene settato se interpretando gli operandi come unsigned (come integer) c'è stato un traboccamento. Poiché l' istruzione SBB gestisce un prestito che può essere stato richiesto in una precedente istruzione, essa può essere usata per scrivere routine che sottraggono numeri più lunghi di 16 bit.

FLAG: **Tutti i flag vengono modificati.**

Operandi	Esempi
Memoria, Registro	SBB DS:2000H, DX
Registro, Memoria	SBB CL, ES:1024H
Registro, Registro	SBB AX, DX
Memoria, Immediato	SBB BYTE OPERAND DS:[DI], 5BH
Registro, Immediato	SBB AX, 54A3H

DECREMENT

FORMATO: **DEC** *destination*

AZIONE: Modifica l'operando destinatario sottraendovi 1; l'operando può essere interpretato sia come unsigned che come integer. Il flag OF viene settato se interpretando l'operando come integer c'è stato un traboccamento; il flag CF non viene modificato.

FLAG: Vengono modificati i flag OF, PF, SF e ZF.

Operandi	Esempi
Memoria	DEC BYTE OPERAND DS:[DI]
Registro	DEC CX

NEGATE

FORMATO: **NEG** *destination*

AZIONE: Interpreta l'operando destinatario come un integer e lo sostituisce con il suo opposto. Qualora l'operazione non sia possibile, viene settato il flag OF. Il flag CF é sempre settato eccetto quando l'operando é zero, nel qual caso viene resettato.

FLAG: Tutti i flag vengono modificati.

Operandi	Esempi
Memoria	NEG BYTE OPERAND DS:[DI]
Registro	NEG CX

COMPARE

FORMATO: **CMP** *destination, source*

AZIONE: Verifica se l'operando destinatario è maggiore, uguale o minore dell'operando sorgente, sia interpretando tali operandi come unsigned che come integer. Nessuno dei due operandi viene modificato mentre i flag vengono aggiornati tenendo conto del risultato della verifica. L'esatto algoritmo di aggiornamento dei flag è noioso da descriversi e non immediato da comprendersi: l'aggiornamento è comunque consistente con l'interpretazione che darà dei flag l'istruzione di salto condizionato o di loop (vedi avanti) che in un programma sensato segue sempre l'istruzione CMP.

FLAG: Tutti i flag vengono modificati.

Operandi	Esempi
Memoria, Registro	CMP DS:2000H, DX
Registro, Memoria	CMP CL, ES:1024H
Registro, Registro	CMP AX, DX
Memoria, Immediato	CMP BYTE OPERAND DS:[DI], 5BH
Registro, Immediato	CMP AX, 54A3H

MULTIPLY

FORMATO: **MUL** *source*

AZIONE: Effettua una moltiplicazione tra un operando sorgente ad 8 bit ed il contenuto del registro AL ovvero tra un operando sorgente a 16 bit ed il contenuto del registro AX, interpretando questi operandi come unsigned. Il risultato della moltiplicazione viene costruito su un numero di bit doppio rispetto a quello degli operandi, cosicché non si ha mai traboccamento. Se il moltiplicando ed il moltiplicatore sono ad 8 bit, allora il risultato viene messo nell'accumulatore AX. Se il moltiplicando ed il moltiplicatore sono a 16 bit, allora il risultato viene messo nell'accumulatore esteso DX_AX. Se la metà più significativa del risultato (in AH o in DX a seconda dei casi) indica che il risultato stesso non sarebbe compattabile su un numero di bit pari a quello degli operandi, allora vengono settati i flag CF e OF.

FLAG: Vengono modificati i flag CF, OF, SF e ZF.

Operandi	Esempi
Memoria	MUL BYTE OPERAND DS:[SI]
Registro	MUL CX

INTEGER MULTIPLY

FORMATO: **IMUL** *source*

AZIONE: Effettua una moltiplicazione tra un operando sorgente ad 8 bit ed il contenuto del registro AL ovvero tra un operando sorgente a 16 bit ed il contenuto del registro AX, interpretando questi operandi come integer. Il risultato della moltiplicazione viene costruito su un numero di bit doppio rispetto a quello degli operandi, cosicché non si ha mai traboccamento. Se il moltiplicando ed il moltiplicatore sono ad 8 bit, allora il risultato viene messo nell'accumulatore AX. Se il moltiplicando ed il moltiplicatore sono a 16 bit, allora il risultato viene messo nell'accumulatore esteso DX_AX. Se la metà più significativa del risultato (in AH o in DX a seconda dei casi) indica che il risultato stesso non sarebbe compattabile su un numero di bit pari a quello degli operandi, allora vengono settati i flag CF e OF.

FLAG: Vengono modificati i flag CF, OF, SF e ZF.

Operandi	Esempi
Memoria	IMUL BYTE OPERAND DS:[SI]
Registro	IMUL CX

DIVIDE

FORMATO: **DIV** *source*

AZIONE: Divide il contenuto dell'accumulatore AX ovvero dell'accumulatore esteso DX_AX per l'operando sorgente a seconda che questi sia ad 8 o a 16 bit, interpretando dividendo e divisore come unsigned. Il quoziente ed il resto vengono costruiti su un numero di bit pari a quello del dividendo e memorizzati rispettivamente in AL ed AH ovvero in AX e DX. Poiché l'istruzione tenta di costruire un quoziente con un numero di bit pari a quello del dividendo, possono verificarsi casi di traboccamento: quando ciò accade, i flag CF ed OF vengono settati ed i contenuti finali di AL ed AH ovvero di AX e DX sono privi di significato.

FLAG: **Vengono modificati i flag CF, OF, SF e ZF.**

Operandi	Esempi
Memoria	DIV BYTE OPERAND DS:[SI]
Registro	DIV CX

INTEGER DIVIDE

FORMATO: **IDIV** *source*

AZIONE: Divide il contenuto dell'accumulatore AX ovvero dell'accumulatore esteso DX_AX per l'operando sorgente a seconda che questi sia ad 8 o a 16 bit, interpretando dividendo e divisore come integer. Il quoziente ed il resto vengono costruiti su un numero di bit pari a quello del dividendo e memorizzati rispettivamente in AL ed AH ovvero in AX e DX. Poiché l'istruzione tenta di costruire un quoziente con un numero di bit pari a quello del dividendo, possono verificarsi casi di traboccamento: quando ciò accade, i flag CF ed OF vengono settati ed i contenuti finali di AL ed AH ovvero di AX e DX sono privi di significato.

FLAG: **Vengono modificati i flag CF, OF, SF e ZF.**

Operandi	Esempi
Memoria	IDIV BYTE OPERAND DS:[SI]
Registro	IDIV CX

CONVERT BYTE TO WORD

FORMATO: **CBW**

AZIONE: Memorizza nell'accumulatore AX un integer a 16 bit avente lo stesso valore del contenuto di AL, interpretato anche esso come integer.

FLAG: Non viene modificato alcun flag.

CONVERT WORD TO DOUBLEWORD

FORMATO: **CWD**

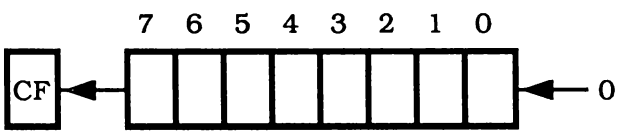
AZIONE: Memorizza nell'accumulatore esteso DX_AX un integer a 32 bit avente lo stesso valore del contenuto di AX, interpretato anche esso come integer.

FLAG: Non viene modificato alcun flag.

SHIFT LOGICAL LEFT

FORMATO: **SHL** *destination*, CL
SHL *destination*

AZIONE: Sostituisce ciascun bit dell'operando destinatario con il bit che gli è immediatamente a destra; il bit meno significativo è sostituito da 0; il bit più significativo viene copiato nel flag CF. Il contenuto del registro CL è interpretato come un unsigned, sia esso n, e l'operazione viene automaticamente ripetuta n volte. Nel caso l'istruzione abbia il secondo formato, il valore di n è assunto per default pari ad 1. Da un punto di vista aritmetico, tale istruzione sostituisce l'operando destinatario (interpretato come unsigned) con il suo prodotto per 2^n: il risultato è sicuramente attendibile solo nel caso n=1 e CF resettato.



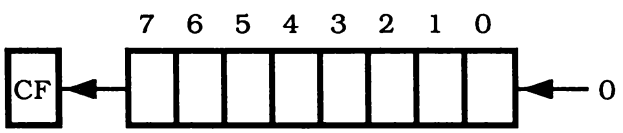
FLAG: **Vengono modificati i flag CF, PF, SF e ZF.**

Operandi	Esempi
Memoria, CL	SHL WORD OPERAND DS:[DI], CL
Registro, CL	SHL BX, CL
Memoria	SHL BYTE OPERAND DS:2000H
Registro	SHL AX

SHIFT ARITHMETIC LEFT

FORMATO: **SAL** *destination*, CL
SAL *destination*

AZIONE: Sostituisce ciascun bit dell'operando destinatario con il bit che gli è immediatamente a destra; il bit meno significativo è sostituito da 0; il bit più significativo viene copiato nel flag CF. Il contenuto del registro CL è interpretato come un unsigned, sia esso n, e l'operazione viene automaticamente ripetuta n volte. Nel caso l'istruzione abbia il secondo formato, il valore di n è assunto per default pari ad 1. Qualora il bit più significativo abbia cambiato anche una sola volta il suo valore originario, allora il flag OF viene settato. Da un punto di vista aritmetico, tale istruzione sostituisce l'operando destinatario (interpretato come integer) con il suo prodotto per 2^n: il risultato è attendibile solo se OF è resettato.



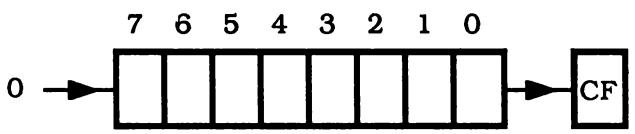
FLAG: Tutti i flag vengono modificati.

Operandi	Esempi
Memoria, CL	SAL WORD OPERAND DS:[DI], CL
Registro, CL	SAL BX, CL
Memoria	SAL BYTE OPERAND DS:2000H
Registro	SAL AX

SHIFT LOGICAL RIGHT

FORMATO: **SHR** *destination*, CL
SHR *destination*

AZIONE: Sostituisce ciascun bit dell'operando destinatario con il bit che gli è immediatamente a sinistra; il bit più significativo è sostituito da 0; il bit meno significativo viene copiato nel flag CF. Il contenuto del registro CL è interpretato come un unsigned, sia esso n, e l'operazione viene automaticamente ripetuta n volte. Da un punto di vista aritmetico, tale istruzione interpreta l'operando destinatario come un unsigned, lo divide per 2^n e lo sostituisce con il quoziente (approssimato per difetto).



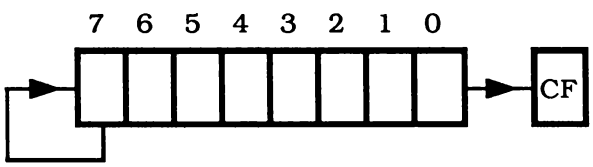
FLAG: **Vengono modificati i flag CF, PF, SF e ZF.**

Operandi	Esempi
Memoria, CL	SHR WORD OPERAND DS:[DI], CL
Registro, CL	SHR BX, CL
Memoria	SHR BYTE OPERAND DS:2000H
Registro	SHR AX

SHIFT ARITHMETIC RIGHT

FORMATO: **SAR** *destination*, CL
SAR *destination*

AZIONE: Sostituisce ciascun bit dell'operando destinatario con il bit che gli è immediatamente a sinistra; il bit più significativo non viene modificato; il bit meno significativo viene copiato nel flag CF. Il contenuto del registro CL è interpretato come un unsigned, sia esso n, e l'operazione viene automaticamente ripetuta n volte. Nel caso l'istruzione abbia il secondo formato, il valore di n è assunto per default pari ad 1. Il flag OF viene sempre resettato. Da un punto di vista aritmetico, tale istruzione interpreta l'operando destinatario come un integer, lo divide per 2^n e lo sostituisce con il quoziente (approssimato per difetto).



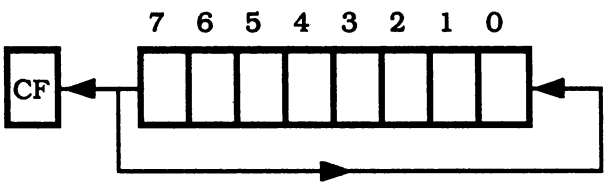
FLAG: **Tutti i flag vengono modificati.**

Operandi	Esempi
Memoria, CL	SAR WORD OPERAND DS:[DI], CL
Registro, CL	SAR BX, CL
Memoria	SAR BYTE OPERAND DS:2000H
Registro	SAR AX

ROTATE LEFT

FORMATO: **ROL** *destination*, CL
ROL *destination*

AZIONE: Sostituisce ciascun bit dell'operando destinatario con il bit che gli è immediatamente a destra; il bit meno significativo è sostituito dal più significativo; il bit più significativo viene anche copiato nel flag CF. Il contenuto del registro CL è interpretato come un unsigned, sia esso n, e l'operazione viene automaticamente ripetuta n volte. Nel caso l'istruzione abbia il secondo formato, il valore di n è assunto per default pari ad 1.



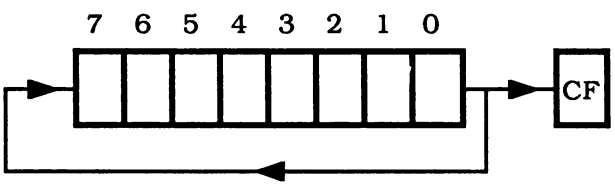
FLAG: Viene modificato il flag CF.

Operandi	Esempi
Memoria, CL	ROL WORD OPERAND DS:[DI], CL
Registro, CL	ROL BX, CL
Memoria	ROL BYTE OPERAND DS:2000H
Registro	ROL AX

ROTATE RIGHT

FORMATO: **ROR** *destination*, CL
ROR *destination*

AZIONE: Sostituisce ciascun bit dell'operando destinatario con il bit che gli è immediatamente a sinistra; il bit più significativo è sostituito dal meno significativo; il bit meno significativo viene anche copiato nel flag CF. Il contenuto del registro CL è interpretato come un unsigned, sia esso n, e l'operazione viene automaticamente ripetuta n volte. Nel caso l'istruzione abbia il secondo formato, il valore di n è assunto per default pari ad 1.



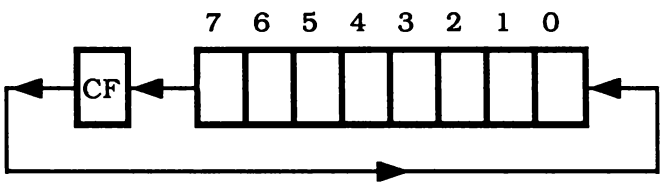
FLAG: Viene modificato il flag CF.

Operandi	Esempi
Memoria, CL	ROR WORD OPERAND DS:[DI], CL
Registro, CL	ROR BX, CL
Memoria	ROR BYTE OPERAND DS:2000H
Registro	ROR AX

ROTATE THROUGH CARRY LEFT

FORMATO: **RCL** *destination*, CL
RCL *destination*

AZIONE: Sostituisce ciascun bit dell'operando destinatario con il bit che gli è immediatamente a destra; il bit meno significativo è sostituito dal valore del flag CF; il bit più significativo viene copiato nel flag CF. Il contenuto del registro CL è interpretato come un unsigned, sia esso n, e l'operazione viene automaticamente ripetuta n volte. Nel caso l'istruzione abbia il secondo formato, il valore di n è assunto per default pari ad 1.



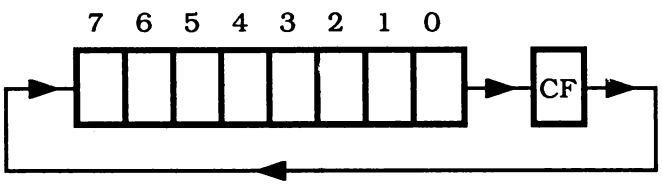
FLAG: **Viene modificato il flag CF.**

Operandi	Esempi
Memoria, CL	RCL WORD OPERAND DS:[DI], CL
Registro, CL	RCL BX, CL
Memoria	RCL BYTE OPERAND DS:2000H
Registro	RCL AX

ROTATE THROUGH CARRY RIGHT

FORMATO: **RCR** *destination*, CL
RCR *destination*

AZIONE: Sostituisce ciascun bit dell'operando destinatario con il bit che gli è immediatamente a sinistra; il bit più significativo è sostituito dal valore del flag CF; il bit meno significativo viene copiato nel flag CF. Il contenuto del registro CL è interpretato come un unsigned, sia esso n, e l'operazione viene automaticamente ripetuta n volte. Nel caso l'istruzione abbia il secondo formato, il valore di n è assunto per default pari ad 1.



FLAG: **Viene modificato il flag CF.**

Operandi	Esempi
Memoria, CL	RCR WORD OPERAND DS:[DI], CL
Registro, CL	RCR BX, CL
Memoria	RCR BYTE OPERAND DS:2000H
Registro	RCR AX

NOT

FORMATO: **NOT** *destination*

AZIONE: Modifica l'operando destinatario applicando a ciascuno dei suoi bit l'operazione logica not.

FLAG: Non viene modificato alcun flag.

Operando	Esempi
Memoria	NOT WORD OPERAND DS:[SI]
Registro	NOT CL

AND

FORMATO: **AND** *destination, source*

AZIONE: Sostituisce ciascun bit dell'operando destinatario con il risultato dell'operazione logica and tra il bit stesso ed il corrispondente bit dell'operando sorgente. Resetta CF ed OF.

FLAG: Tutti i flag vengono modificati.

Operandi	Esempi
Memoria, Registro	AND DS:2000H, DX
Registro, Memoria	AND CL, ES:1024H
Registro, Registro	AND AX, DX
Memoria, Immediato	AND BYTE OPERAND DS:[DI], 5BH
Registro, Immediato	AND AX, 54A3H

OR

FORMATO: **OR** *destination, source*

AZIONE: Sostituisce ciascun bit dell'operando destinatario con il risultato dell'operazione logica or tra il bit stesso ed il corrispondente bit dell'operando sorgente. Resetta CF ed OF.

FLAG: Tutti i flag vengono modificati.

Operandi	Esempi
Memoria, Registro	OR DS:2000H, DX
Registro, Memoria	OR CL, ES:1024H
Registro, Registro	OR AX, DX
Memoria, Immediato	OR BYTE OPERAND DS:[DI], 5BH
Registro, Immediato	OR AX, 54A3H

XOR

FORMATO: **XOR** *destination, source*

AZIONE: Sostituisce ciascun bit dell'operando destinatario con il risultato dell'operazione logica exclusive or tra il bit stesso ed il corrispondente bit dell'operando sorgente. Resetta CF ed OF.

FLAG: Tutti i flag vengono modificati.

Operandi	Esempi
Memoria, Registro	XOR DS:2000H, DX
Registro, Memoria	XOR CL, ES:1024H
Registro, Registro	XOR AX, DX
Memoria, Immediato	XOR BYTE OPERAND DS:[DI], 5BH
Registro, Immediato	XOR AX, 54A3H

TEST

FORMATO: **TEST** *destination, source*

AZIONE: Setta i flag in accordo al risultato ottenuto applicando l'operazione logica and tra ciascun bit dell'operando destinatario ed il corrispondente bit dell'operando sorgente; non vengono modificati nè l'operando sorgente nè l'operando destinatario. In un programma sensato, tale istruzione è seguita da una istruzione di salto condizionato o di loop (vedi oltre). Resetta CF ed OF.

FLAG: **Tutti i flag vengono modificati.**

Operandi	Esempi
Memoria, Registro	TEST DS:2000H, DX
Registro, Memoria	TEST CL, ES:1024H
Registro, Registro	TEST AX, DX
Memoria, Immediato	TEST BYTE OPERAND DS:[DI], 5BH
Registro, Immediato	TEST AX, 54A3H

JUMP NEAR

FORMATO: **JMP** CS:*displacement*
JMP CS:[*register*]
JMP VECTORED NEAR DS:*displacement*
JMP VECTORED NEAR ES:*displacement*

AZIONE: Immette nel registro IP il displacement specificato nell'istruzione (primo formato), il contenuto del registro specificato nell'istruzione (secondo formato) ovvero il contenuto di due locazioni di memoria, l'indirizzo della prima delle quali è specificato nell'istruzione in accordo alla modalità di indirizzamento diretto (terzo e quarto formato). In ogni caso il "flusso del programma" rimane all'interno del segmento codice corrente.

FLAG: Non viene modificato alcun flag.

JUMP FAR

FORMATO: **JMP** *selector:displacement*
JMP VECTORED FAR DS:*displacement*
JMP VECTORED FAR ES:*displacement*

AZIONE: Immette nel registro IP il displacement specificato nell'istruzione (primo formato) ovvero il contenuto di due locazioni di memoria, l'indirizzo della prima delle quali è specificato nell'istruzione in accordo alla modalità di indirizzamento diretto (secondo e terzo formato); immette in CS il selettore specificato nell'istruzione (primo formato) ovvero il contenuto delle due locazioni di memoria consecutive a quelle utilizzate per rinnovare il contenuto di IP (secondo e terzo formato). In ogni caso il flusso del programma non è vincolato a rimanere all'interno del segmento codice corrente.

FLAG: Non viene modificato alcun flag.

ISTRUZIONI CONDITIONAL JUMP

FORMATO: **Jcondition** CS:*displacement*

AZIONE: Se un esame dei flag indica che la condizione **condition** è soddisfatta, allora tali istruzioni immettono nel registro IP il displacement che esse stesse contengono; in caso contrario non modificano il contenuto di IP. Una di queste istruzioni non esamina i flag, bensì il contenuto di CX. In ogni caso il "flusso del programma" rimane all'interno del segmento codice corrente. Di seguito sono riassunti i codici operativi delle istruzioni di salto condizionato e, per ciascuno di essi, è spiegato brevemente il significato della pertinente condizione.

JE	Segue una istruzione CMP: il salto avviene se l'operando destinatario era uguale all'operando sorgente.
JNE	segue una istruzione CMP: il salto avviene se l'operando destinatario non era uguale all'operando sorgente.
JA	segue una istruzione CMP: il salto avviene se l'operando destinatario era maggiore dell'operando sorgente, essendo gli operandi interpretati come unsigned.
JAE	segue una istruzione CMP: il salto avviene se l'operando destinatario era maggiore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati come unsigned.
JB	segue una istruzione CMP: il salto avviene se l'operando destinatario era minore dell'operando sorgente, essendo gli operandi interpretati come unsigned.
JBE	segue una istruzione CMP: il salto avviene se l'operando destinatario era minore od uguale rispetto all'operando sorgente, essendo gli operandi interpretati come unsigned.

JG	segue una istruzione CMP: il salto avviene se l'operando destinatario era maggiore dell'operando sorgente, essendo gli operandi interpretati come integer.
JGE	segue una istruzione CMP; il salto avviene se l'operando destinatario era maggiore o uguale rispetto allo operando sorgente, essendo gli operandi interpretati come integer.
JL	segue una istruzione CMP: il salto avviene se l'operando destinatario era minore dell'operando sorgente, essendo gli operandi interpretati come integer.
JLE	segue una istruzione CMP: il salto avviene se l'operando destinatario era minore od uguale rispetto all'operando sorgente, essendo gli operandi interpretati come integer.
JZ	il salto avviene se il risultato dell'istruzione precedente è stato zero.
JNZ	il salto avviene se il risultato dell'istruzione precedente è stato diverso da zero.
JC	il salto avviene se l'istruzione precedente ha settato il flag CF.
JNC	il salto avviene se l'istruzione precedente ha resettato il flag CF.
JO	il salto avviene se l'istruzione precedente ha settato il flag OF.
JNO	il salto avviene se l'istruzione precedente ha resettato il flag OF.
JS	il salto avviene se l'istruzione precedente ha settato il flag SF.
JNS	il salto avviene se l'istruzione precedente ha resettato il flag SF.
JPE	il salto avviene se l'istruzione precedente ha settato il flag PF.
JPO	il salto avviene se l'istruzione precedente ha resettato il flag PF.
JCXZ	il salto avviene se il contenuto di CX è zero.

FLAG: Non viene modificato alcun flag.

ISTRUZIONI LOOP

FORMATO: **LOOPcondition** CS:*displacement*

AZIONE: Decrementano il contenuto del registro CX senza modificare i flag; di poi se il contenuto di CX è ancora diverso da zero e se un esame dei flag indica che la condizione **condition** è soddisfatta, allora immettono nel registro IP il displacement che esse stesse contengono; in caso contrario non modificano il contenuto di IP. In ogni caso il "flusso del programma" rimane all'interno del segmento codice corrente. Queste istruzioni sono usate per ripetere un pacchetto di istruzioni al più n volte (dove n è il numero unsigned precedentemente memorizzato in CX) e comunque finché la condizione specificata rimane vera. Di seguito sono riassunti i codici operativi delle istruzioni di loop e, per ciascuno di essi, è spiegato brevemente il significato della pertinente condizione:

LOOP	Il salto avviene se, dopo l'autodecremento di CX, il suo contenuto è ancora diverso da zero (nessuna ulteriore condizione è testata).
LOOPE	Segue una istruzione CMP: il salto avviene se, dopo l'autodecremento di CX, il suo contenuto è ancora diverso da zero e se l'operando destinatario trattato dalla istruzione CMP era uguale all'operando sorgente.
LOOPNE	Segue una istruzione CMP: il salto avviene se, dopo l'autodecremento di CX, il suo contenuto è ancora diverso da zero e se l'operando destinatario trattato dalla istruzione CMP era diverso rispetto all'operando sorgente.

LOOPZ	Il salto avviene se, dopo l'autodecremento di CX, il suo contenuto è ancora diverso da zero e se il risultato dell'istruzione precedente è stato zero.
LOOPNZ	Il salto avviene se, dopo l'autodecremento di CX, il suo contenuto è ancora diverso da zero e se il risultato dell'istruzione precedente è stato diverso da zero.

FLAG: Non viene modificato alcun flag.

CALL NEAR

FORMATO: **CALL** CS:*displacement*
CALL CS:[*register*]
CALL VECTORED NEAR DS:*displacement*
CALL VECTORED NEAR ES:*displacement*

AZIONE: Salva nella pila corrente il contenuto del registro IP e di poi modifica il contenuto di tale registro in modo del tutto simile a come fa l'istruzione jump near. Il flusso del programma rimane all'interno del segmento codice corrente; la pila conserva l'offset dell'istruzione di rientro (cioè dell'istruzione che nel programma seguiva l'istruzione CALL)

FLAG: Non viene modificato alcun flag.

CALL FAR

FORMATO: **CALL** *selector:displacement*
CALL VECTORED FAR DS:*displacement*
CALL VECTORED FAR ES:*displacement*

AZIONE: Salva nella pila corrente il contenuto dei registri IP e CS e di poi modifica il contenuto di tali registri in modo del tutto simile a come fa l'istruzione jump far. Il flusso del programma non è vincolato a rimanere all'interno del segmento codice corrente; la pila conserva l'indirizzo dell'istruzione di rientro (cioè dell'istruzione che nel programma seguiva l'istruzione CALL)

FLAG: Non viene modificato alcun flag.

RETURN NEAR

FORMATO: **RET/N**

AZIONE: Rimuove una parola dalla pila e con essa rinnova il contenuto di IP. Tale istruzione deve chiudere un pacchetto di istruzioni (sottoprogramma), la prima delle quali sia atta ad essere messa in esecuzione da una CALL near; il suo scopo è quello di far sì che sia rimessa in esecuzione la pertinente istruzione di rientro.

FLAG: Non viene modificato alcun flag.

RETURN FAR

FORMATO: **RET/F**

AZIONE: Rimuove due parole dalla pila e con essa rinnova il contenuto di IP e di CS. Tale istruzione deve chiudere un pacchetto di istruzioni (sottoprogramma), la prima delle quali sia atta ad essere messa in esecuzione da una CALL far; il suo scopo è quello di far sì che sia rimessa in esecuzione la pertinente istruzione di rientro.

FLAG: Non viene modificato alcun flag.

NO OPERATION

FORMATO: **NOP**

AZIONE: Provoca un azione nulla da parte del processore.

FLAG: Non viene modificato alcun flag.

HALT

FORMATO: **HLT**

AZIONE: Provoca la cessazione di ogni attività da parte del processore

FLAG: Non viene modificato alcun flag.

IL LINGUAGGIO ASSEMBLER PER IL MICROPROCESSORE mE86

1. GENERALITA'

Il linguaggio assembler è una estensione del linguaggio mnemonico rispetto al quale ha il vantaggio di permettere una descrizione completa del programma sia per quanto concerne i segmenti contenenti istruzioni (segmenti codice) che i segmenti contenenti operandi (segmenti dati e segmenti stack): il linguaggio assembler mette inoltre a disposizione un simbolismo molto leggibile per specificare le modalità di indirizzamento.

L'assemblatore per il microprocessore mE86 deve essere un assemblatore assoluto che accetti un unico file contenente il programma in linguaggio assembler (*file sorgente* o di tipo *.asm*) e generi un file contenente codice e operandi già pronti per essere immessi nello spazio di memoria del microprocessore (*file eseguibile* o di tipo *.exe*): non sono infatti definite direttive per l'assemblaggio separato ed il successivo collegamento di più moduli di programma contenuti in file sorgenti distinti.

La struttura di un file eseguibile è riportata in Fig. 1. Per ogni segmento viene specificato il selettore, l'offset della prima locazione utilizzata, il numero di locazioni costituenti il segmento (o lunghezza del segmento in byte) ed il contenuto di tali locazioni (o corpo del segmento).

Gli ultimi 32 bit del file rappresentano lo entry_point, cioè l'indirizzo logico della prima istruzione eseguendo la quale il processore sarà in grado di eseguire correttamente tutte le altre.

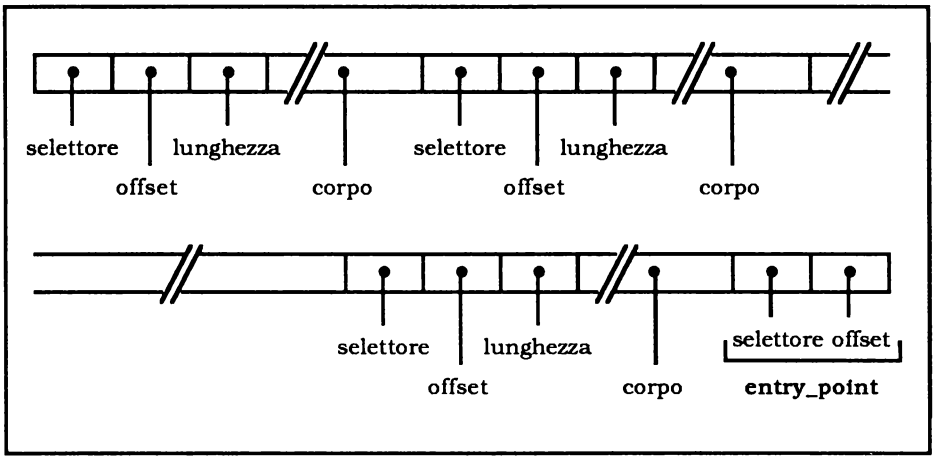


Fig. 1 - Struttura di un file eseguibile.

2. STRUTTURA DI UN PROGRAMMA ASSEMBLER

La struttura di principio in linguaggio assembler di un programma costituito da un segmento stack, da N segmenti dati e da M segmenti codice è riportata nella Tabella 1 (le principali parole chiave sono scritte in caratteri maiuscoli; i simboli specificabili dall'utente a suo gradimento, in caratteri minuscoli).

Le parole chiave `SEGMENT`, `ENDS` (END Segment) e `END` costituiscono alcune delle *direttive* per l'assemblatore e servono come "punteggiatura": in particolare la direttiva `END` indica all'assemblatore che il programma è da considerarsi completo. I simboli `mystack`, `mydata1`, ..., `mydataN`, `mycode1`, ..., `mycodeM` permettono di riferire i segmenti stessi senza conoscerne i selettori, ma indicando tali selettori come `SELECTOR mystack`, `SELECTOR mydata1`, ..., `SELECTOR mydataN`, `SELECTOR mycode1`, ..., `SELECTOR mycodeM`. Qualora il programmatore intenda stabilire che il selettore del segmento `mydata1` sia ad esempio `3BA5H`, la pertinente direttiva `SEGMENT` diviene

```
mydata1     SEGMENT SELECTED BY 3BA5H
```

```
mystack     SEGMENT STACK
            corpo del segmento
mystack     ENDS
mydata1     SEGMENT
            corpo del segmento
mydata1     ENDS
            .
            .
            .
mydataN     SEGMENT
            corpo del segmento
mydataN     ENDS
mycode1     SEGMENT
            ASSUME CS:mycode1, SS:mystack
            ASSUME DS:mydatah, ES:mydatak
            corpo del segmento
mycode1     ENDS
            .
            .
            .
mycodeM     SEGMENT
            ASSUME CS:mycodeM, SS:mystack
            ASSUME DS:mydatar, ES:mydatas
            corpo del segmento
mycodeM     SEGMENT

            END entry_point
```

Tabella 1 - Struttura di un programma assembler.

La parola chiave STACK serve solo per ricordare al programmatore quale segmento andrà usato come stack. Il modo di specificare lo *entry_point* sarà chiarito tra poco. La direttiva ASSUME che precede il corpo di ogni segmento codice indica all assemblatore i quattro segmenti che saranno visti come segmenti correnti durante l'esecuzione delle istruzioni costituenti il corpo del segmento stesso (nell'esempio deve essere h, k, r, s $\in$ (1, 2 ,...., N) ed h $\neq$ k e r $\neq$ s). Tale direttiva in parte serve all'assemblatore per un corretto svolgimento del suo compito di produzione di file eseguibili, in parte è

una ridondanza che comunque l'assemblatore vuole sia soddisfatta. Più precisamente, l'indicazione CS:... è sostanzialmente inutile, ma obbligatoria; le indicazioni DS:..., ES:... e SS:... possono risultare effettivamente necessarie all'assemblatore (la loro omissione, quando necessarie, causa una segnalazione di errore).

3. COME DICHIARARE IL CORPO DI UN SEGMENTO DATI

Il corpo di un segmento dati è dichiarabile mediante due direttive, note come Define Byte DB e Define Word DW. Per chiarire il loro uso, supponiamo che un segmento dati sia stato dichiarato in un file sorgente come segue:

```
mydata      SEGMENT SELECTED BY 2000H
alphab      DB 2AH
betab       DB (?)
gammab      DB "ms"
deltab      DB 2AH,"m"
omegab      DB 2 DUP 3BH
tetab       DB 2 DUP (?)
alphaw      DW 2A5FH
betaw       DW (?)
omegaw      DW 2 DUP A1B3H
tetaw       DW 2 DUP (?)
mydata      ENDS
```

Se andiamo ad analizzare il file eseguibile prodotto dall'assemblatore, troviamo come controparte del segmento sopra descritto la sequenza :

```
.|2000|0000|0018|2A??6D732A6D3B3B????2A5F????A1B3A1B3????????|.
```

Se ne deduce che il segmento di selettore 2000H è definito a partire dalla locazione di offset 0000H ed il suo corpo è di 0016H (22 decimale) byte.

La prima direttiva

```
alphab      DB 2AH
```

definisce come 2AH il primo byte del corpo del segmento. La seconda direttiva

```
betab       DB (?)
```

definisce come non-specificato (??) il secondo byte del corpo del segmento. La terza direttiva

```
gammab      DB "ms"
```

definisce come appartenente al corpo del segmento la ulteriore sequenza di byte 6D73H, cioè la sequenza delle codifiche ASCII dei caratteri tra apici. La quarta direttiva

```
deltab      DB 2AH,"m"
```

indica che il corpo del segmento deve contenere un ulteriore byte pari a 2AH, seguito dalla codifica ASCII 6DH del carattere tra apici. La quinta direttiva

```
omegab      DB 2 DUP 3BH
```

definisce, come appartenenti al corpo del segmento, 2 ulteriori byte entrambi pari a 3BH: la parola chiave DUP sta per DUPlicate. La sesta direttiva

```
tetab       DB 2 DUP (?)
```

definisce, come appartenenti al corpo del segmento, 2 ulteriori byte non-specificati. La settima direttiva

```
alphaw      DW 2A5FH
```

definisce come appartenente al corpo del segmento la word 2A5FH. L'ottava direttiva

```
betaw       DW (?)
```

definisce, come appartenente al corpo del segmento, una word non-specificata (????). La nona direttiva

```
omegaw      DW 2 DUP A1B3H
```

definisce come appartenenti al corpo del segmento 2 word, ciascuna pari a A1B3H. Infine, la decima direttiva

```
tetaw       DW 2 DUP (?)
```

definisce come appartenenti al corpo del segmento 2 word non-specificate.

Le etichette alphab, betab, gammab, deltab, omegab, tetab, alphaw, betaw, omegaw e tetaw che precedono le parole chiave DB e DW sono opzionali ed utilizzabili per riferire in modo molto leggibile le locazioni del segmento (come sarà illustrato tra poco). Si dice anche che una direttiva del tipo

```
alphab      DB .....
```

definisce una *variabile di tipo byte*, di *nome simbolico* alphab, di *selettore* SELECTOR alphab e di *offset* OFFSET alphab. Similmente una direttiva del tipo

```
alphaw      DW .....
```

definisce una *variabile di tipo word.*, di *nome simbolico* alphaw, di *selettore* SELECTOR alphaw e di *offset* OFFSET alphaw. Una notazione equivalente per dichiarare variabili fa uso della direttiva LABEL da utilizzarsi tenendo presente che:

```
alphab      LABEL BYTE
            DB .....
```

equivale a

```
alphab      DB .....
```

mentre

```
alphaw      LABEL WORD
            DW ......
```

equivale a

```
alphaw      DW .....
```

4. COME DICHIARARE LO SPAZIO DI I/O

La struttura dello spazio di I/O non è influenzabile dal programmatore essendo invece legata a quali, a quante ed a come le interfacce sono state utilizzate per supportare fisicamente (un piccolo sottoinsieme del) le porte di tale spazio. A livello di linguaggio assembler, il programmatore può dichiarare quali porte intende riferire ed assegnare ad esse un identificatore simbolico più comodo da usarsi rispetto all'offset numerico. Supposto ad esempio che nello spazio di I/O siano supportate da opportuni circuiti integrati la porta di offset 003FH e la coppia di porte contigue di offset 0A51H e 0A52H accessibili come una unica porta a 16 bit, allora un programmatore che voglia utilizzare queste porte può dichiarare tale volontà come segue:

```
IOSPACE          DECLARATION
device_status    BPORT AT 003FH
device_buffer    WPORT AT 0A51H
IOSPACE          ENDD
```

La direttiva

```
device_status    BPORT AT 003FH
```

definisce a livello di linguaggio assembler una *porta di tipo byte*, di *nome simbolico* `device_status` e con *offset* `OFFSET device_status` pari a 003FH; la direttiva

```
device_buffer    WPORT AT 0A51H
```

definisce a livello di linguaggio assembler una *porta di tipo word*, di *nome simbolico* `device_buffer` e con *offset* `OFFSET device_buffer` pari a 0A51H.

I nomi simbolici potranno essere utilizzati per riferire le porte stesse come sarà specificato poco sotto.

Anche se può apparire scontato, è bene rimarcare che nessuna traccia esiste nel file eseguibile, in corrispondenza alle dichiarazioni di porta presenti nel file sorgente.

5. COME DICHIARARE IL CORPO DI UN SEGMENTO CODICE

Il corpo di un segmento codice ha, a livello di linguaggio assembler, una struttura del tipo:

```
mycode      SEGMENT
            ASSUME CS:mycode, DS:...., ES:...., SS:....
sott1       PROC NEAR
            istruzione
            ..........
            ..........
            RET
sott1       ENDP
sott2       PROC FAR
            istruzione
            ..........
            ..........
            ..........
            istruzione
            RET
sott2       ENDP
            istruzione
            ..........
            ..........
            ..........
            istruzione
mydata      ENDS
```

La coppia di direttive PROC, ENDP racchiude un blocco di istruzioni costituenti un *sottoprogramma* (PROCedure): se la direttiva PROC è accompagnata dalla parola chiave NEAR, allora tale sottoprogramma è detto di *tipo near* ed è chiamabile solo tramite una istruzione CALL che appartenga al segmento mycode; se invece la direttiva PROC è accompagnata dalla parola chiave FAR,

allora tale sottoprogramma è detto di *tipo far* ed è chiamabile anche tramite una istruzione `CALL` che appartenga ad un altro segmento.

Come scrivere in linguaggio assembler le istruzioni di chiamata a sottoprogramma

L'etichetta che precede la direttiva `PROC` può essere utilizzata per chiamare il sottoprogramma anche senza conoscere il selettore e l'offset della prima istruzione del sottoprogramma stesso. Ad esempio, l'istruzione

```
CALL sott1
```

appartenente al segmento `mycode`, è considerata dall'assemblatore equivalente a

```
CALL CS:OFFSET sott1
```

L'assemblatore calcola quindi l'offset della prima istruzione del sottoprogramma di etichetta `sott1` e traduce in linguaggio macchina l'istruzione in linguaggio mnemonico ottenuta dopo tale calcolo.

Similmente, una istruzione del tipo

```
CALL sott2
```

appartenente al segmento `mycode` o ad un altro segmento codice, è considerata dall'assemblatore equivalente a

```
CALL SELECTOR sott2:OFFSET sott2
```

L'assemblatore calcola quindi il selettore e l'offset della prima istruzione del sottoprogramma di etichetta `sott2` e traduce in linguaggio macchina l'istruzione in linguaggio mnemonico ottenuta dopo tale calcolo.

Si noti infine come l'istruzione `RET` non necessita, come nel linguaggio mnemonico, della parola chiave `NEAR` o `FAR` essendo implicito, dalla sua collocazione, se trattasi di `RET/N` o di `RET/F`.

Come scrivere in linguaggio assembler le istruzioni di salto

Per avere nelle istruzioni di salto una leggibilità paragonabile a quella ottenuta nella istruzione CALL, si può assegnare un *etichetta* ad una istruzione ed usare tale etichetta per specificare un salto all'istruzione stessa. Ad esempio, scrivendo

```
            ..........
            ..........
istr1       LABEL NEAR
            MOV AX, DX
            ..........
            ..........
istr2       LABEL FAR
            INC CX
            ..........
            ..........
```

si assegna all'istruzione MOV AX,DX l'etichetta istr1 di *tipo near.* e all'istruzione INC CX l'etichetta istr2 di *tipo far.* Le due istruzioni sono riferibili in un istruzione di salto appartenente allo stesso segmento codice, scrivendo rispettivamente

```
JMP istr1
```

e

```
JMP istr2
```

La seconda istruzione è riferibile in una istruzione di salto appartenente a qualunque segmento codice, scrivendo semplicemente

```
JMP istr2
```

In entrambi i casi, l'assemblatore è in grado di calcolare gli indirizzi logici delle due istruzioni etichettate, di trasformare le istruzioni di salto in modo da renderle sintatticamente corrette nel linguaggio mnemonico e di poi tradurle in linguaggio macchina.

Una etichetta di tipo near può essere specificata in modo più semplice: ad esempio lo scrivere

```
istr1:      MOV AX, DX
```

è considerato dall'assemblatore equivalente allo scrivere

```
istr1 LABEL NEAR
            MOV AX, DX
```

Come specificare lo entry_point

Gli ultimi 32 bit del file rappresentano lo entry_point, cioè l'indirizzo logico della prima istruzione eseguendo la quale il processore sarà in grado di eseguire correttamente tutte le altre. A livello di linguaggio assembler lo entry_point va dichiarato dando una etichetta, sia `start`, alla istruzione che si vuole sia eseguita per prima, e chiudendo il file sorgente con la direttiva

```
            END start
```

Tale direttiva spinge l'assemblatore a calcolare l'indirizzo completo dell'istruzione etichettata `start` e a chiudere con tale indirizzo il file eseguibile.

Come indirizzare in linguaggio assembler le variabili

L'uso delle variabili semplifica notevolmente le modalità di indirizzamento ed aumenta la leggibilità del programma. Alcuni esempi chiariranno come ciò sia possibile. Sia

```
mydata      SEGMENT
            ........
            ........
betab       DB (?)
betaw       DW (?)
            ........
mydata      ENDS
```

un segmento dati espresso in linguaggio assembler. Il selettore di tale segmento è ignoto al programmatore: esso può tuttavia riferirlo indicandolo come SELECTOR mydata ovvero come SELECTOR betab ovvero come SELECTOR betaw. Anche gli offset delle variabili all'interno del segmento non sono noti al programmatore: esso può tuttavia riferirli indicandoli come OFFSET betab ed OFFSET betaw rispettivamente. Pertanto l'istruzione

```
MOV AX, SELECTOR mydata
```

spinge l'assemblatore a calcolare il selettore del segmento mydata e a tradurre in linguaggio macchina la istruzione in linguaggio mnemonico ottenuta dopo tale calcolo: in altre parole nell'istruzione MOV l'operando sorgente è specificato secondo la modalità di indirizzamento immediato. Un altro esempio di indirizzamento immediato dell'operando sorgente è

```
MOV DX, OFFSET betab
```

Un caso abbastanza diverso è rappresentato dalla istruzione, valida nel linguaggio assembler

```
MOV AL, betab
```

L'assemblatore va a guardare la direttiva ASSUME all'inizio della della descrizione del segmento codice e se essa è ad esempio del tipo

```
ASSUME ..... , ES:mydata, .....
```

allora l'istruzione MOV è considerata equivalente a

```
MOV AL, ES:OFFSET betab
```

L'assemblatore calcola pertanto l'offset della variabile betab e traduce in linguaggio macchina la istruzione in linguaggio mnemonico ottenuta dopo tale calcolo: in altre parole nell'istruzione MOV l'operando sorgente è specificato secondo la modalità di indirizzamento diretto.

Come altro esempio consideriamo l'istruzione, valida nel linguaggio assembler:

```
MOV AL, betab[SI]
```

L'assemblatore va a guardare la direttiva ASSUME all'inizio della descrizione del segmento codice e se essa è ad esempio del tipo

```
ASSUME ..... , ES:mydata, .....
```

allora l'istruzione MOV è considerata equivalente a

```
MOV AL, ES:OFFSET betab[DI]
```

L'assemblatore calcola pertanto l'offset della variabile betab e traduce in linguaggio macchina la istruzione in linguaggio mnemonico ottenuta dopo tale calcolo: in altre parole nell'istruzione MOV l'operando sorgente è specificato secondo la modalità di indirizzamento indiciato. Come altro esempio consideriamo l'istruzione, valida nel linguaggio assembler:

```
INC betaw
```

Apparentemente mancano le parole chiave BYTE OPERAND o WORD OPERAND: in realtà esse non sono necessarie in quanto la variabile betaw è, grazie alla dichiarazione che la definisce, di tipo word. Pertanto l'assemblatore considera l'istruzione precedente equivalente alla istruzione (anche essa valida nel linguaggio assembler)

```
INC WORD OPERAND betaw
```

Se di poi la direttiva ASSUME all'inizio della descrizione del segmento codice ha la forma

```
ASSUME ..... , ES:mydata, .....
```

allora l'assemblatore considera l'istruzione INC equivalente a

```
INC WORD OPERAND ES:OFFSET betaw
```

e pertanto calcola l'offset della variabile `betaw` e traduce in linguaggio macchina la istruzione in linguaggio mnemonico ottenuta dopo tale calcolo.

Considerazioni simili fanno capire che istruzioni del tipo

```
MOV AX, betab
MOV AL, betaw
```

sono scorrette non essendovi compatibilità fra l'operando sorgente e quello destinatario.

Come indirizzare in linguaggio assembler le porte

Consideriamo le porte di nome simbolico `device_status` e `device_buffer` introdotte precedentemente. I seguenti esempi mostrano come esse possono essere riferite a livello di linguaggio assembler. L'istruzione

```
IN AL, device_status
```

è considerata dall'assemblatore equivalente all'istruzione

```
IN AL, IO:OFFSET device_buffer
```

e quindi all'istruzione

```
IN AL, IO:003FH
```

e come tale è tradotta in linguaggio macchina. Similmente l'istruzione

```
MOV DX, OFFSET device_buffer
```

è considerata dall'assemblatore equivalente all'istruzione

```
MOV DX, 0A51H
```

e come tale è tradotta in linguaggio macchina.

6. LA DIRETTIVA EQUATE ED IL SUO USO

Una delle direttive di uso più comune è la direttiva EQU (EQUate) che permette di assegnare ad una costante un nome simbolico. Ad esempio, la direttiva

```
enne        EQU 30H
```

spinge l'assemblatore a sostituire il simbolo enne con 30H, ovunque esso sia nello spezzone di programma che segue la direttiva stessa. Il poter utilizzare il simbolo enne al posto della costante 30H permette una maggiore leggibilità, ma soprattutto una maggiore manutenibilità del programma stesso: qualora ad esempio si voglia sostituire la costante 30H con la costante 1BH basta apportare un solo cambiamento nella direttiva EQU a prescindere dal numero di volte che il simbolo enne è utilizzato. Notiamo che per evitare possibili ambiguità a livello di tipo di operandi (a 8 o 16 bit) è opportuno a volte far seguire la parola chiave EQU dalle consuete parole chiave BYTE OPERAND o WORD OPERAND.

La direttiva EQU può essere utilizzata in modo più sofisticato di quanto detto fino ad ora. Ad esempio

```
beta        EQU WORD OPERAND ES:2000H
```

definisce una *variabile di tipo word*, di *nome simbolico* beta, di *selettore specificato da* ES ed di *offset* 2000H. Pertanto

```
MOV AX, enne
MOV AX, beta
```

pur avendo apparentemente la stessa forma, prevedono rispettivamente per l'operando sorgente un indirizzamento di tipo immediato e di tipo diretto.

7. I COMMENTI

E` possibile inserire commenti in una linea conte-

nente una istruzione o una direttiva oppure definire una intera linea di commenti, facendo precedere i caratteri ASCII costituenti il commento dalla parola chiave ";".

ASPETTI FISICI DEL MICROPROCESSORE mE86

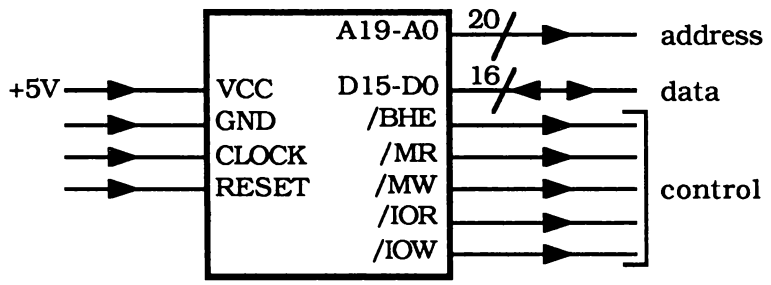
1. PIN DI COLLEGAMENTO

La piedinatura del microprocessore mE86 e la funzione dei principali pin sono descritti nella Fig. 1.

Tutti i 20 pin per gli indirizzi fisici sono usati durante i cicli di accesso allo spazio lineare di memoria fisica, mentre solo 16 di essi (A15-A0) sono utilizzati durante i cicli di accesso allo spazio di I/O: questo in accordo a quanto già detto, e cioè che la memoria fisica ha una capcità di 1 Mbyte e lo spazio di I/O di 64 Kbyte.

I pin per i dati sono 16. Gli 8 pin D7-D0 (o pin bassi o pin pari) sono utilizzati per trasferire byte da/a locazioni di indirizzo fisico (e quindi di offset logico) pari; gli 8 pin D15-D8 (o pin alti o pin dispari) sono utilizzati per trasferire byte da/a locazioni di indirizzo fisico (e quindi di offset logico) dispari. L'intero insieme di pin è utilizzabile per trasferire in un solo ciclo due byte (questo è molto utile durante la fase di fetch ed ogni volta che nella fase di esecuzione gli operandi sono a 16 bit) purchè la locazione sorgente/destinatario del byte meno significativo abbia indirizzo fisico (e quindi di offset logico) pari. La coppia di bit sui due pin /BHE (Bus High Enable) ed A0 (pin per il bit meno significativo degli indirizzi) codifica infatti il tipo di trasferimento, in accordo alla seguente legge:

/BHE	A0	Numero di bit trasferiti
0	0	16 sull'intero bus
0	1	8 sul bus alto
1	0	8 sul bus basso
1	1	non specificato



RESET	ingresso	**Reset**: una transizione da 0 a 1 porta il microprocessore nello stato iniziale con CS a FFFF e DS, ES, SS, IP a 0000.
A19-A0	uscita	**Address**: supportano i bit di indirizzo fisico durante le operazioni di lettura e di scrittura sia nello spazio di memoria che nello spazio di I/O; rimangono flottanti durante le operazioni interne.
D15-D0	bidirezionali	**Data**: supportano i bit costituenti i dati durante le operazioni di lettura e di scrittura sia nello spazio di memoria che nello spazio di I/O; rimangono flottanti durante le operazioni interne.
/BHE	uscita	**Bus High Enable**: durante le operazioni di lettura e scrittura indica se la parte alta del bus dati (pin D15-D8) è o non è impegnata.
/MR	uscita	**Memory Read**: un impulso su tale pin indica che il processore compie una operazione di lettura nello spazio di memoria.
/MW	uscita	**Memory Write**: un impulso su tale pin indica che il processore compie una operazione di scrittura nello spazio di memoria.
/IOR	uscita	**I/O Read**: un impulso su tale pin indica che il processore compie una operazione di lettura nello spazio di I/O.
/IOW	uscita	**I/O Write**: un impulso su tale pin indica che il processore compie una operazione di scrittura nello spazio di I/O.

Fig 1 - Piedinatura del microprocessore mE86.

Da quanto detto sul modo di accedere agli operandi, deriva che il programmatore può liberamente specificare offset pari o offset dispari: tuttavia, se per un operando a 16 bit viene specificato un offset di partenza dispari, il tempo di accesso si raddoppia rispetto al caso di offset pari in quanto i 16 bit vengono trasferiti in due distinti cicli.

2. ARCHITETTURA DI UN SEMPLICE CALCOLATORE BASATO SUL MICROPROCESSORE mE86

In Fig. 2 è riportato uno schema di principio di un calcolatore basato sul microprocessore mE86.

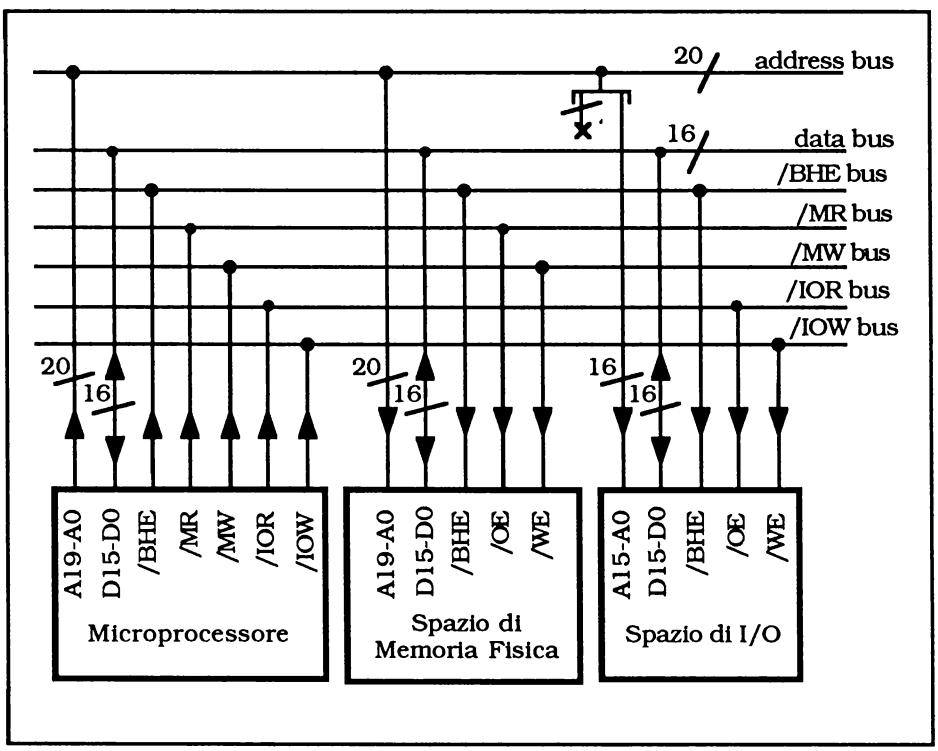


Fig. 2 - Schema di un calcolatore basato sul microprocessore mE86.

La piedinatura esterna del complesso delle circuiterie che implementano in tutto od in parte gli spazi lineari di memoria fisica e di I/O debbono essere consistenti con la piedinatura del microprocessore.

L'insieme dei fili che connettono microprocessore, spazio di memoria fisica e spazio di I/O è detto *bus*. I pin /OE ed /WE di ingresso agli spazi di memoria fisica e di I/O servono a recepire gli impulsi di comando lettura e scrittura rispettivamente.

L'evoluzione temporale dei livelli logici nel bus è riportata in modo qualitativo in Fig. 3.

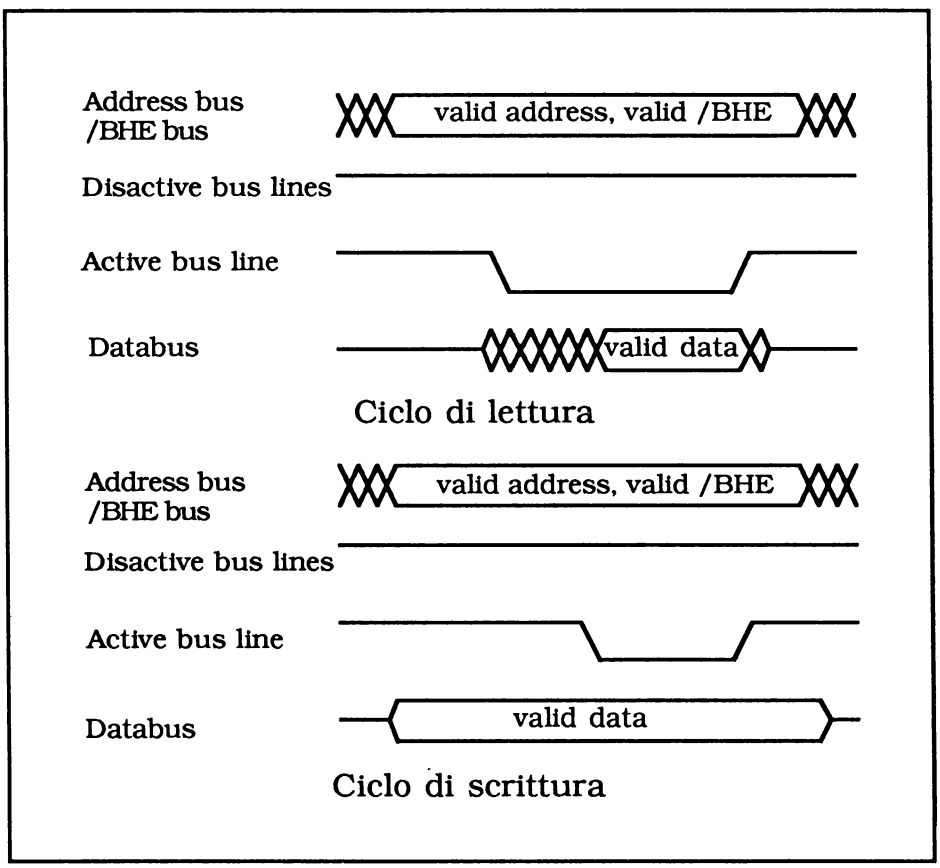


Fig. 3 - Evoluzione temporale dei livelli logici nel bus.

Delle quattro linee /MR bus, /MW bus, /IOR bus e /IOW bus , tre sono tenute a livello logico 1 durante ciascun ciclo (disactive bus lines); una è invece percorsa da un impulso emesso dal microprocessore (active bus line). Più precisamente:

- durante un ciclo di lettura nello spazio di memoria, l'impulso viene emesso tramite il pin /MR, mentre sui pin /MW, /IOR ed /IOW è mantenuto costantemente il livello logico 1.
- durante un ciclo di lettura nello spazio di I/O l'impulso viene emesso tramite il pin /IOR, mentre sui pin /MR, /MW, ed /IOW è mantenuto costantemente il livello logico 1;
- durante un ciclo di scrittura nello spazio di memoria, l'impulso viene emesso tramite il pin /MW, mentre sui pin /MR, /IOR ed /IOW è mantenuto costantemente il livello logico 1;
- durante un ciclo di scrittura nello spazio di I/O l'impulso viene emesso tramite il pin /IOW, mentre sui pin /MR, /MW, ed /IOR è mantenuto costantemente il livello logico 1.

3. ORGANIZZAZIONE DELLA MEMORIA FISICA E DELLO SPAZIO DI I/O

Lo spazio lineare di memoria fisica ha capacità totale di 1 Mbyte ed è diviso in due banchi da 512 Kbyte ciascuno, come mostrato in Fig. 4.

Un banco è selezionato se il bit su A0 è 0, l'altro se il bit su /BHE è 0: il primo banco o banco pari viene quindi a fornire le locazioni di indirizzo fisico (e quindi di offset logico) pari ; l'altro banco o banco dispari, le locazioni di indirizzo fisico (e quindi di offset logico) dispari.

Una organizzazione simile è valida anche per lo spazio di I/O, anche se molte volte uno stesso integrato può fornire locazioni (porte nella terminologia corrente) sia al banco pari che a quello dispari ed impe-

dire di fatto la lettura e la scrittura di word in un unico ciclo.

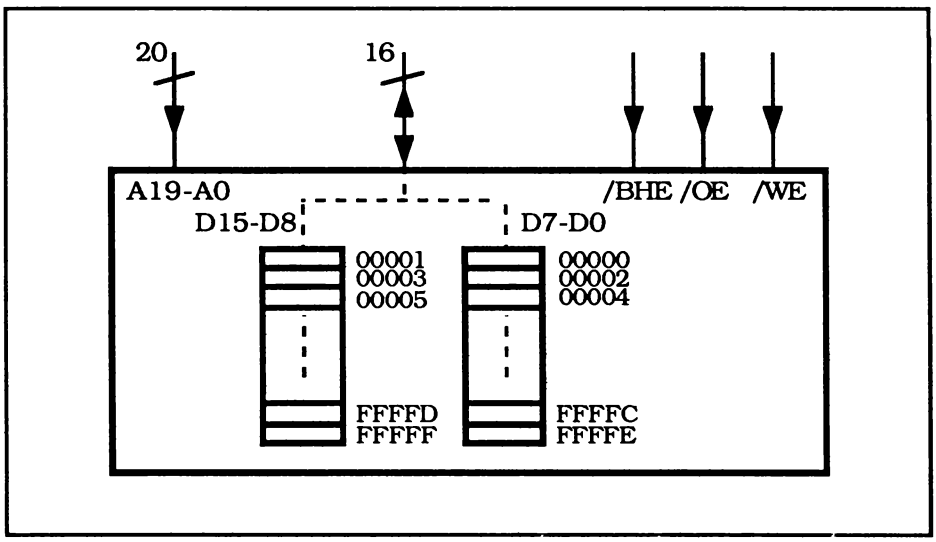


Fig. 4 - I due banchi costituenti lamemoria fisica.

L'INTERFACCIA SERIALE EART

1. GENERALITA`

Una interfaccia è un dispositivo che da un lato colloquia con un microprocessore comportandosi come una piccola memoria e dall'altro con un apparato esterno (trasduttore) secondo protocolli (in genere) standard. L'integrato EART è una interfaccia con capacità di ricevere e trasmettere secondo lo standard per Comunicazioni Seriali Asincrone EIARS232C: essa è pertanto atta ad essere connessa ad apparati esterni quali terminali video, modem, altre interfacce seriali asincrone etc.

Una comunicazione seriale tra un dispositivo trasmettitore ed un dispositivo ricevitore avviene utilizzando un unico collegamento, sul quale viaggiano serialmente i singoli bit costituenti l'informazione da scambiare. I bit vengono inviati a gruppi (trame nella terminologia corrente): all'interno della trama i bit sono trasmessi ad intervalli di tempo regolari, mentre trame successive possono essere separate da un qualunque intervallo di tempo. Ogni bit richiede un certo tempo T per essere trasmesso (tempo di bit). Il numero di bit inviati nell'unità di tempo (se la trasmissione fosse continua) si dice baud-rate, e vale ovviamente 1/T. A seconda della tecnica di trasmissione usata, il baud-rate va da qualche decina di bit/sec a qualche decina di Mbit/sec. Entrando nel merito di come un trasmettitore ed un ricevitore si sincronizzano per un corretto scambio di dati, si ha una ulteriore classificazione di una comunicazione seriale in comunicazione seriale asincrona e comunicazione seriale sincrona.

In una comunicazione seriale asincrona le trame

sono costituite da un numero di bit estremamente limitato, da 7 a 12, e ciascuna trasporta una informazione utile da 5 a 8 bit, detta comunemente carattere. Più precisamente, una trama comprende:

- un bit di start,
- un carattere (da 5 a 8 bit),
- un eventuale bit di parità,
- alcuni bit di stop (1 oppure 2).

Tra una trama e l'altra la linea viene mantenuta nello stato di riposo (marking). Il bit di start corrisponde a portare la linea nello stato attivo (space), mentre ogni bit di stop corrisponde a tenere la linea nello stato di marking. Ogni bit di informazione (come pure il bit di parità) è trasmesso tenendo la linea nello stato marking o space a seconda che valga 1 oppure 0 (codifica NRZ). In Fig. 1 è mostrato l'andamento del segnale sulla linea per una trama con 7 bit per carattere, un bit di parità pari ed un bit di stop: nello standard EIA RS232C comunemente usato, lo stato space corrisponde a +10V e lo stato marking a -10V. Valori tipici di baud-rate sono: 110, 300, 600, 1200, 2400, 4800, 9600, 19200 bit/sec.

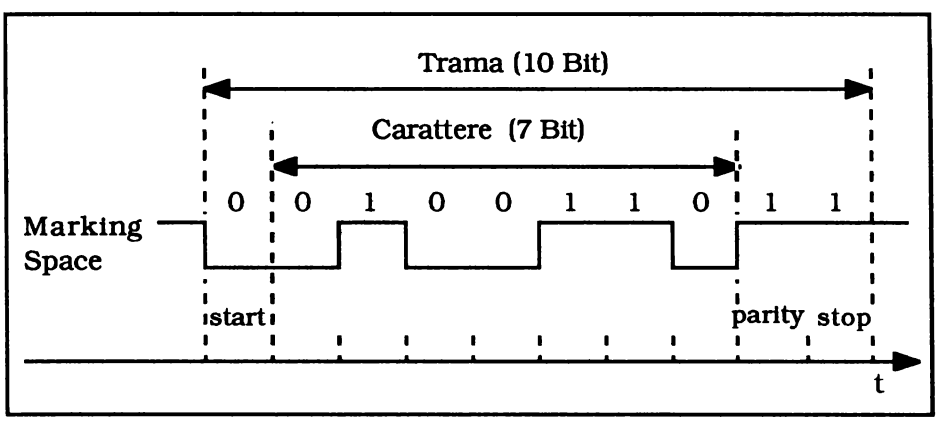


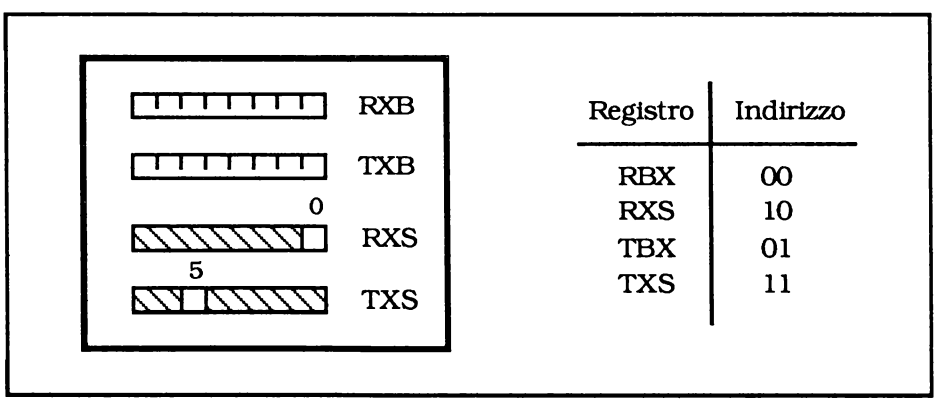
Fig. 1 - Andamento del segnale sulla linea relativamente ad una trama.

L'interfaccia EART (Educational Asynchronous Receiver Transmitter) è in grado supportare una comunicazione asincrona seriale full-duplex essendo dotata di due circuiterie, una atta a fungere da ricevitore e l'altra da trasmettitore rispetto ad un trasduttore esterno. Le caratteristiche della comunicazione seriale sia in ricezione che in trasmissione sono:

baud-rate: 9600 bit/sec
bit per carattere: 8
parità: disabilitata
bit di stop: 1

2. MODELLO FUNZIONALE DELL'INTERFACCIA EART

Da un punto di vista funzionale, l'interfaccia EART appare come un insieme di 4 porte (dette anche registri) da 8 bit (vedi Fig. 2).



Registro	Indirizzo
RBX	00
RXS	10
TBX	01
TXS	11

Fig. 2 - Modello funzionale dell'interfaccia EART.

Il registro RXB (Receiver Buffer) serve per memorizzare un byte, proveniente dal trasduttore esterno, fino a chè il byte stesso non viene prelevato dal microprocessore. Il registro TXB (Transmitter Buffer) serve per memorizzare un byte, proveniente dal micropro-

cessore, per il tempo necessario affinchè sia spedito al trasduttore esterno. La funzione principale dei registri RXS e TXS (Status Registers) deriva dalle seguenti considerazioni.

L'invio di informazioni (generalmente sequenze di byte) da un trasduttore esterno al microcalcolatore comporta un trasferimento di byte dal trasduttore esterno al registro RXB dell' interfaccia, e un trasferimento di byte dal registro RXB dell'interfaccia al microprocessore. Similmente l'invio di informazioni (generalmente sequenze di byte) da un microcalcolatore ad un trasduttore esterno comporta un trasferimento di byte dal microprocessore al registro TXB dell' interfaccia, e un trasferimento di byte dal registro TXB dell'interfaccia al dispositivo esterno.

Il microprocessore mE86 usa il registro AL per scambiare byte con i registri delle interfaccie montate nello spazio di I/O (il trasferimento di un byte da un registro dell'interfaccia al registro AL del microprocessore e viceversa avvengono mediante l'esecuzione di istruzioni IN ed OUT).

I trasferimenti di byte dal trasduttore esterno al registro RXB del'interfaccia e dal registro TXB dell'interfaccia al trasduttore esterno avvengono invece via hardware, ad opera della interfaccia stessa. Più precisamente (vedi Fig. 2), in ciascuno dei due registri di stato RXS e TXS è presente un bit indicatore di pronto (*flag* nella terminologia corrente): il bit presente nel registro RXS è detto *flag di buffer di ingresso pieno* ed indica (quando vale 1) che un nuovo byte è pervenuto alla interfaccia dal trasduttore e che tale byte è disponibile per il microprocessore nel registro RXB; il bit presente nel registro TXS è detto *flag di buffer di uscita vuoto* ed indica (quando vale 1) che il byte precedentemente immesso dal microprocessore nel registro TXB è stato spedito al trasduttore. Pertanto, se il flag di buffer di ingresso vale 1, il microprocessore può correttamente prelevare un nuovo byte dal registro RXB dell'interfaccia: quando tale prelievo avviene (esecu-

zione di una istruzione di ingresso), l'interfaccia resetta automaticamente il flag ed è pronta a ricevere un nuovo byte dal trasduttore. Similmente, se il flag di buffer di uscita vuoto vale 1, il microprocessore può immettere un nuovo byte nel registro TXB dell'interfaccia: quando tale immissione avviene (esecuzione di una istruzione di uscita), l'interfaccia resetta automaticamente il flag ed inizia a trasferire il byte al trasduttore. La gestione di una operazione di ingresso o di uscita a controllo di programma comporta di verificare, per ogni byte da trasferire, lo stato dei flag.

Un sottoprogramma di ingresso che porta nel registro AL del microprocessore un nuovo byte prelevandolo dal registro RXB dell'interfaccia, può quindi essere strutturato come segue:

```
INPUT       PROC FAR
TEST1:      IN AL,RXS
            AND AL,01H  ;esame del bit n.0 di RXS
            JZ TEST1
            IN AL,RXB   ;prelievo del byte
            RET
INPUT       ENDP
```

In modo del tutto analogo, un sottoprogramma per far uscire (tramite il registro TXB) un byte, preventivamente posto dal programma principale nel registro AL, può essere strutturato nel seguente modo.

```
OUTPUT      PROC FAR
            PUSH AX
TEST2:      IN AL,TXS
            AND AL,20H  ;esame del bit n.5 di TXS
            JZ TEST2
            POP AX
            OUT TXB,AL  ;emissione del byte
            RET
OUTPUT      ENDP
```

Ovviamente tali sottoprogrammi debbono essere preceduti in un programma dalla direttive per dichia-

rare porte: se l'interfaccia EART è montata nello spazio di I/O all'interno del banco in modo tale che i suoi registri vengano ad avere i seguenti offset

Registro	Offset
RXB	00F8H
RXS	00FCH
TXB	00FAH
TXS	00FEH

allora le direttive da inserire nel programma sono:

```
IOSPACE     DECLARATION
RXB         BPORT AT 00F8H
RXS         BPORT AT 00FCH
TXB         BPORT AT 00FAH
TXS         BPORT AT 00FEH
IOSPACE     ENDD
```

3. CONFIGURAZIONE FISICA DELL'INTERFACCIA EART

La piedinatura dell'integrato EART e la funzione dei vari pin sono riportate nella Fig. 3 (per semplicità sono omessi i pin relativi alla alimentazione, massa e clock).

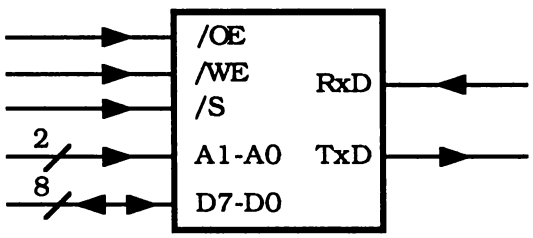
L'interfaccia può essere montata nello spazio di I/O del microprocessore mE86 secondo lo schema di Fig. 4. I pin per dati dell'interfaccia sono collegati al bus basso, cosicchè gli offset generati dal microprocessore per accedere ai registri dell'interfaccia devono essere pari: tali offset si ottengono, a partire dai corrispondenti indirizzi interni dei registri, applicando la regola:

$$\text{offset} = \text{offset_base} + (2 \cdot \text{indirizzo_interno})$$

dove lo *offset_base* è legato alla struttura del riconoscitore di indirizzo.

Supponendo ad esempio di voler montare l'interfaccia a partire dallo offset_base 00F8H, il riconoscitore di indirizzo deve produrre una uscita bassa quando i bit sulle linee A15-A3 sono 0000 0000 1111 1 ed il bit

sulla linea A0 è 0.



A1-A0	ingresso	**Address**: supportano i bit di indirizzo per i registri RXB, RXS, TXB, TXS durante le operazioni di lettura e di scrittura.
D7-D0	bidirezionali	**Data**: supportano i bit costituenti i dati durante le operazioni di lettura e di scrittura.
/OE	ingresso	**Output enable**: un impulso su tale pin è interpretato dal chip come la richiesta di una operazione di lettura.
/WE	ingresso	**Write enable**: un impulso su tale pin è interpretato dal chip come la richiesta di una operazione di scrittura.
/S	ingresso	**Chip select**: un livello logico 0 su tale pin predispone il chip a rispondere a richieste di operazioni di lettura o di scrittura.
RxD	ingresso	**Serial input**: tramite questo pin vengono raccolti in sequenza i bit costituenti il nuovo contenuto del registro RXB.
TxD	uscita	**Serial output**: attraverso tale pin vengono emessi in sequenza i bit contenuti nel registro TXB.

Fig. 3 - Piedinatura dell'integrato EART.

Dal lato trasduttore (pin RxD e TxD), l'interfaccia abbisogna di una coppia di amplificatori del tipo TMS 75189 e TMS 75188 per adattarsi agli standard elettrici EIARS232C.

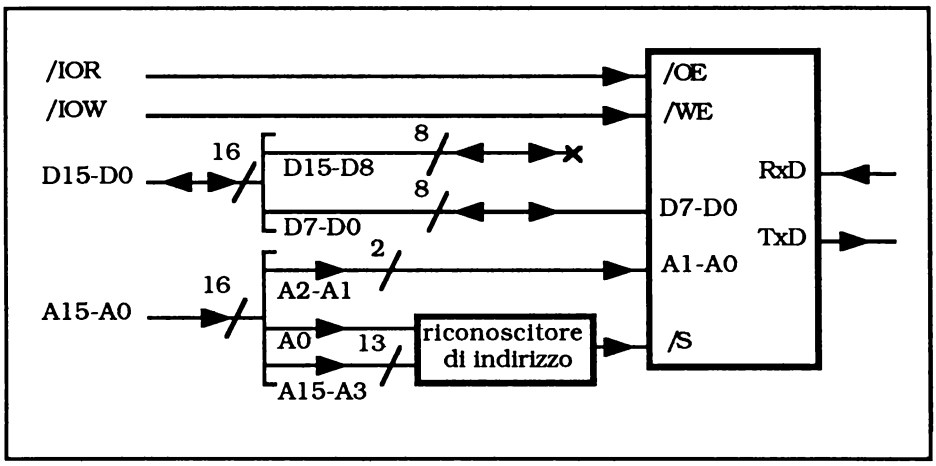


Fig. 4 - Schema di montaggio nello spazio di I/O per l'interfaccia EART .

IL CALCOLATORE SU SINGOLA SCHEDA EB86 ED IL SUO AMBIENTE DI SVILUPPO

1. IL CALCOLATORE SU SINGOLA SCHEDA EB86

Il calcolatore su singola scheda EB86 è un calcolatore didattico basato sul microprocessore mE86. La memoria fisica è costituita da un insieme di integrati EPROM e RAM; nello spazio di I/O è montata la sola interfaccia seriale EART. Un programma monitor (il cui codice è residente su EPROM) gestisce il calcolatore e ad esso sono riservati i segmenti con selettore inferiore a 1000H e superiore a FFFEH. La memoria RAM disponibile per i programmi utente ha una capacità di 256 Kbyte. Gli offset nello spazio di I/O dei registri dell'interfaccia sono dati dalla seguente tabella:

Registro	Offset
RXB	00F8H
RXS	00FCH
TXB	00FAH
TXS	00FEH

2. AMBIENTE DI SVILUPPO PER IL CALCOLATORE EB86

Il calcolatore EB86 non possiede direttamente gli strumenti (editore, assemblatore e linker) per sviluppare software. Tali strumenti sono disponibili invece su un calcolatore *host* a cui il calcolatore EB86 è connesso in qualità di *target* tramite la linea seriale gestita dall'interfaccia EART come illustrato in Fig. 1. Il calcolatore host deve essere gestito dal Sistema Operativo

DOS 3.30 oppure dal Sistema Operativo UNIX V.

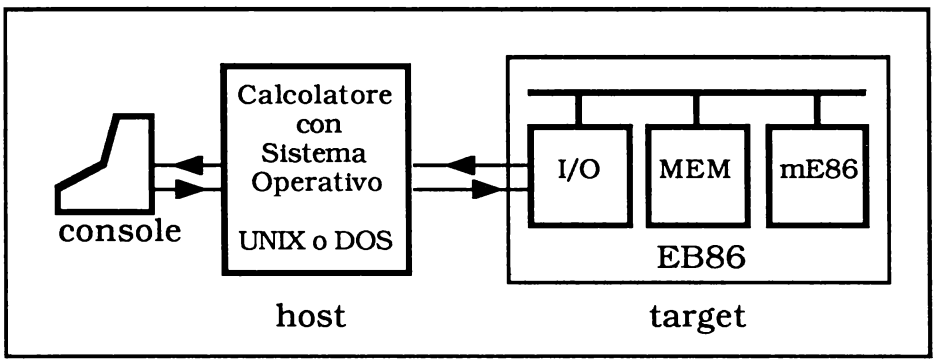


Fig. 1 - Ambiente di sviluppo per il calcolatore EB86.

L'ambiente di sviluppo su host è interattivo e si presenta mostrando il seguente menù di servizi:

```
1) EDITAZIONE di un file sorgente
2) ASSEMBLAGGIO ASSOLUTO di un file sorgente
3) VISUALIZZAZIONE e STAMPA di file

4) INVIO al calcolatore TARGET di un file eseguibile
5) CONNESSIONE diretta con il calcolatore TARGET

6) ELENCO dei file sorgente presenti su HOST
7) DUPLICAZIONE su HOST di un file sorgente
8) CANCELLAZIONE su HOST di un file sorgente

0) FINE SESSIONE


SCELTA ?
```

Si possono pertanto editare e modificare file sorgenti (scelta n. 1), produrre file eseguibili (scelta n. 2) ed inviarli al target (scelta n. 4), fare elencazioni, duplicazioni, cancellazioni e stampe di file (scelte n. 6, 7, 8 e 3). Il servizio selezionato con la scelta n. 5 consiste nell'aprire una connessione diretta tra il terminale a cui è seduto l'utente (*console*) ed il target (in altre parole il

sistema host si comporta come un cortocircuito): in tal modo un programma precedentemente caricato in memoria può essere messo in esecuzione e collaudato; l'host si inserisce di nuovo tra console e calcolatore target, digitando la parola HOST.

3. L'ASSEMBLATORE SU HOST

L'assemblatore su host è un assemblatore assoluto che accetta file sorgenti scritti nel linguaggio assembler del microprocessore mE86 e produce direttamente file eseguibili. L'assemblatore produce anche file in cui, accanto al programma in linguaggio assembler, è riportato il programma in linguaggio mnemonico (ricordiamo che il linguaggio macchina del microprocessore è tenuto riservato) con in più una serie di informazioni sugli indirizzi delle istruzioni e degli operandi (*file listato* o di tipo *.lst*). In quanto assemblatore assoluto, esso svolge anche le funzioni che normalmente sono di pertinenza del linker e quindi assegna i selettori ai segmenti: nel fare ciò si preoccupa di evitare che i segmenti si sovrappongano in memoria fisica e non assegna selettori minori di 1000H e maggiori di FFFEH dichiarandoli riservati per i segmenti costituenti il programma monitor. L'assemblatore pone inoltre le seguenti limitazioni legate alla sua implementazione:

- ogni simbolo (nome di segmento, etichetta di istruzione, ...) che il programmatore voglia introdurre, deve iniziare con una lettera e non deve essere costituito da più di 10 caratteri alfanumerici; il numero massimo di segmenti che possono costituire un programma è 25; il numero massimo di direttive `EQU` che possono essere introdotte è 500 sotto UNIX e 100 sotto DOS; il numero massimo di simboli che possono essere definiti globalmente è 2000 sotto UNIX e 300 sotto DOS;
- per motivi di protezione l'istruzione `HLT` (HaLT) non è riconosciuta e quindi non è tradotta in linguaggio macchina; è invece definita la direttiva `EXIT` che è considerata

come equivalente alla istruzione

```
JMP 0C00H:0000H
```

essendo `0C00H:0000H` l'indirizzo del ciclo principale del programma monitor.

4. IL MONITOR SU TARGET

Il calcolatore target EB86 è gestito da un programma monitor residente su EPROM che si presenta inviando il prompt **M*** sulla linea seriale connessa all'interfaccia EART (ricordiamo che l'utente può aprire una connessione diretta tra la console ed il calcolatore target optando per la scelta n. 5 del menù; tale connessione è abbattibile digitando la parola HOST). Il monitor, oltre a colloquiare con il sistema host, permette all'utente di mandare in esecuzione e di collaudare un proprio programma preventivamente caricato nella memoria del target stesso. Questo servizio è esplicato dal monitor attraverso l'accettazione dei 7 seguenti ***comandi di debugging.***

I comandi di debugging

I comandi di debugging sono:

```
U     [FROM indirizzo] [FOR numero]
D     [FROM indirizzo] [FOR numero]
EB    FROM indirizzo
EW    FROM indirizzo
R     [registro]
G     [FROM indirizzo] [TO indirizzo]
T     [FROM indirizzo] [FOR numero]
```

Tutto ciò che è indicato tra parantesi quadre può essere omesso, valendo opportune regole di default (le parentesi quadre non vanno digitate); le entità `indirizzo` e `numero` vanno espresse in esadecimale omettendo la classica lettera H finale. Valgono inoltre le seguenti regole:

- parole chiave e parametri possono essere scritti in caratteri minuscoli o maiuscoli o in una loro combinazione;
- l'indirizzo può essere specificato sia come `selector:offset` che come `selector_register:offset`;
- i comandi diventano attivi solo dopo aver premuto su console il tasto `CR` (Ritorno Carrello);
- se viene riscontrato un errore nella sintassi del comando battuto, viene emesso un messaggio di errore;
- il monitor emette messaggi di errore se, nell'eseguire un comando, si verificano situazioni tali che il comando stesso non possa essere completamente espletato;
- il monitor emette messaggi di errore se, mentre il microprocessore esegue uno spezzone di programma, si verificano situazioni anomale.
- qualora si sia perso il controllo del calcolatore target occorre digitare `Ctrl Break` o `Break` sotto UNIX e `Ctrl C` sotto DOS: il target viene resettato e l'host abbatte la connessione diretta console-target.

Il comando `U` (*Unassemble*) mostra sulla console il contenuto di tante locazione di memoria quante ne sono necessarie per contenere un numero di istruzioni pari a quello specificato dopo la parola chiave `FOR`; l'indirizzo della prima locazione ispezionata è quello che segue la parola chiave `FROM`. Con tale comando sono accessibili solo segmenti codice: il contenuto delle locazioni (istruzioni in linguaggio macchina) è presentato in linguaggio mnemonico.

Il comando `D` (*Dump*) mostra sulla console il contenuto di tante locazione di memoria quante ne sono specificate dal numero che segue la parola chiave `FOR`; l'indirizzo della prima locazione ispezionata è quello che segue la parola chiave `FROM`. Con tale comando sono accessibili solo segmenti di tipo dati e stack: il contenuto delle locazioni è presentato in esadecimale (non viene mostrata la lettera H che generalmente indica la base sedici).

I comandi `EB` e `EW` (*Enter Byte* ed *Enter Word*) permettono di modificare il contenuto di una o più locazioni di un segmento di tipo dati o stack; la prima loca-

zione coinvolta è quella il cui indirizzo è specificato nel comando stesso; la differenza fra i due comandi sta nel fatto che il primo agisce su una locazione alla volta, il secondo su coppie di locazioni; i contenuti delle locazioni vengono presentati e vanno inviati in esadecimale. I comandi sono interattivi; ad esempio il comando `EB FROM indirizzo` si presenta visualizzando il contenuto della prima locazione da ispezionare e ponendo una domanda, il tutto in una forma del tipo :

```
indirizzo  OLD VALUE= valore  NEW VALUE?=
```

A questa richiesta (ed a tutte le successive richieste simili) occorre rispondere come segue:

i) digitare quello che si vuole che diventi il nuovo contenuto della locazione ovvero passare direttamente al punto successivo se si vuole che il contenuto della locazione rimanga invariato;
ii) premere un ritorno carrello per uscire dal comando ovvero premre `N` seguito da ritorno carrello per passare ad ispezionare la locazione di indirizzo successivo ovvero premere `P` seguito da ritorno carrello per passare ad ispezionare la locazione di indirizzo precedente.

Il comando `R` (*Register*) mostra sulla console il contenuto dei registri del microprocessore e ne permette la modifica: tali contenuti vengono presentati e vanno inviati in esadecimale. Il comando è interattivo e segue regole abbastanza simili a quelle dei comani `EB` ed `EW`.

Il comando `G` (*Go*) mette in esecuzione un programma a partire dall'istruzione il cui indirizzo segue la parola chiave `FROM` e fa arrestare l'esecuzione quando viene individuata l'istruzione il cui indirizzo (indirizzo di *break point*) segue la parola chiave `TO` (l'istruzione in oggetto non viene eseguita): a questo punto il comando mostra sulla console il contenuto di tutti i registri del microprocessore e il mnemonico della prossima istruzione da eseguire.

Il comando T (*Trace*) mette in esecuzione un programma a partire dall'istruzione il cui indirizzo segue la parola chiave FROM facendo arrestare l'esecuzione dopo tante istruzioni quante ne sono specificate dal numero che segue la parola chiave FOR: per ogni istruzione che viene eseguita, il comando mostra sulla console il contenuto di tutti i registri del microprocessore e il mnemonico della prossima istruzione da eseguire.

Le regole di default sono le seguenti:

- omettere FROM indirizzo nei comandi G e T equivale a specificare come selettore il contenuto attuale di CS e come offset il contenuto attuale di IP;
- omettere FROM indirizzo nei comandi U e D equivale a specificare come indirizzo quello della locazione successiva alla locazione ispezionata per ultima durante lo svolgimento del comando (omonimo) precedente;
- omettere FOR numero nel comando T equivale a specificare FOR 1;
- omettere FOR numero nel comando D equivale a specificare FOR FF;
- omettere FOR numero nel comando U equivale a specificare FOR A;
- omettere TO indirizzo nel comando G equivale a lasciare che il programma messo in esecuzione, segua il suo flusso senza alcun vincolo; se il programma termina normalmente tornando al monitor, appare la scritta

  Program terminated normally
- omettere il nome di un registro nel comando R equivale a far visualizzare il contenuto attuale di tutti i registri senza avere la possibilità di effettuare modifiche.

5. I SOTTOPROGRAMMI DI UTILITA'

Oltre al programma monitor, sono registrati in EPROM anche 10 sottoprogrammi di utilità che un programmatore può invocare mediante l'istruzione

```
CALL 0C00H:offset
```

Il passagio dei parametri avviene attraverso registri generali. Di seguito tutti i programmi di utilità sono descritti in dettaglio.

Uscita di un carattere sulla console

Questa procedura invia al video della console, tramite l'interfaccia EART, il carattere la cui codifica ASCII è nel registro AL.

Parametri passati alla procedura	**Registri**
Codifica ASCII del carattere da emettere	AL
Parametri ritornati al programma chiamante	
Nessuno	
Entry Point 0C00H:0012H	

Ingresso di un carattere da console

Questa procedura attende che l'interfaccia EART riceva la codifica ASCII di un carattere (ciò avverrà quando l'utente avrà premuto un tasto sulla tastiera della console), e porta tale codifica ASCII nel registro AL. Non viene fatta l'eco del carattere sul video della console.

Parametri passati alla procedura	**Registri**
Nessuno	
Parametri ritornati al programma chiamante	
Codifica ASCII del carattere ricevuto	AL
Entry Point 0C00H:001BH	

Uscita di una riga sulla console

Questa procedura invia al video della console le codifiche ASCII contenute in un buffer di memoria: l'ultima codifica ASCII deve essere 0DH (codifica del carattere CR).

Parametri passati alla procedura	**Registri**
Indirizzo logico della prima locazione del buffer	DX, BX
Parametri ritornati al programma chiamante	
Nessuno	
Entry Point: 0C00H:0018H	
Diagnostica: se nel buffer non viene trovata la codifica ASCII del carattere CR, la procedura continua ad inviare al video il contenuto di tutte le locazioni successive a quelle del buffer ed appartenenti allo stesso segmento; non viene segnalato alcun errore	

Uscita di un messaggio sulla console

Questa procedura invia al video della console le codifiche ASCII contenute in un buffer di memoria

Parametri passati alla procedura	**Registri**
Indirizzo logico della prima locazione del buffer	DX, BX
Capacità in byte del buffer	CX
Parametri ritornati al programma chiamante	
Nessuno	
Entry Point: 0C00H:0015H	

Ingresso di una riga da console

Questa procedura immette in un buffer di memoria le codifiche ASCII dei caratteri di una stringa digitata da console e terminata premendo il tasto CR (Carriage Return). Viene inserita nel buffer anche la codifica ASCII del carattere LF (Line Feed). Di ogni carattere viene fatta l'eco sul video della console. Il carattere BS (Backspace) viene interpretato dalla procedura come una direttiva per cancellare dal buffer l'ultimo carattere immessovi.

Parametri passati alla procedura	**Registri**
Indirizzo logico della prima locazione del buffer	DX, BX
Capacità in byte del buffer	CX
Parametri ritornati al programma chiamante	
Nessuno tramite registri	
Entry Point: 0C00H:001EH	
Diagnostica: se la stringa è più lunga della capacità del buffer specificata in CX, essa viene parzialmente perduta	

Conversione da binario ad esadecimale (cifre esadecimali codificate ASCII)

Questa procedura converte un numero binario di 8 bit nel corrispondente numero esadecimale. Il numero binario deve risiedere in AL; le codifiche ASCII delle due cifre esadecimali ottenute dopo la conversione sono lasciate in DX.

Parametri passati alla procedura	**Registri**
Numero binario da convertire	AL
Parametri ritornati al programma chiamante	
Codifica ASCII della cifra esadecimale più sigificativa	DH
Codifica ASCII della cifra esadecimale meno sigificativa	DL
Entry Point 0C00H:000FH	

Conversione da esadecimale a binario (cifre esadecimali codificate ASCII)

Questa procedura converte un numero esadecimale di due cifre nel corrispondente numero binario. Le codifiche ASCII delle cifre esadecimali debbono risiedere in AX, il risultato della conversione è lasciato in DL.

Parametri passati alla procedura	**Registri**
Codifica ASCII della cifra esadecimale più sigificativa	AH
Codifica ASCII della cifra esadecimale meno sigificativa	AL
Parametri ritornati al programma chiamante	
Risultato della conversione in binario	DL
Entry Point: 0C00H:000CH	
Diagnostica: se le codifiche ASCII contenute in AH e AL non corrispondono a cifre esadecimali non viene segnalato alcun errore, ma il risultato è inattendibile	

Conversione da decimale a binario (cifre decimali codificate ASCII)

Questa procedura converte da base dieci a base due nel campo dei numeri unsigned. Le codifiche ASCII delle cifre decimali debbono risiedere in un buffer di memoria: il risultato della conversione è lasciato in AX.

Parametri passati alla procedura	**Registri**
Indirizzo logico della prima locazione del buffer	DX, BX
Numero delle cifre decimali	CX
Parametri ritornati al programma chiamante	
Risultato dalla conversione in binario	AX
Entry Point: 0C00H:0003H	
Diagnostica: se il numero decimale supera 65535 non sono segnalati errori, ma il risultato della conversione è inattendibile.	

Conversione da binario a decimale (cifre decimali codificate ASCII)

Questa procedura converte da base due a base dieci nel campo dei numeri unsigned. Il numero binario da convertire deve trovarsi in AX; la codifica ASCII delle cifre decimali è lasciata in un buffer di memoria. Il programmatore deve specificare il numero di cifre con cui desidera ottenere il risultato. La procedura fa precedere le cifre significative con degli

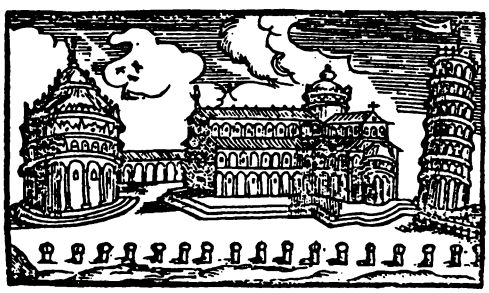
zeri per raggiungere il numero di cifre richiesto.

Parametri passati alla procedura	**Registri**
Numero binario da convertire	AX
Indirizzo logico della prima locazione del buffer	DX, BX
Numero delle cifre decimali del risultato (al più 5)	CX
Parametri ritornati al programma chiamante	
Nessuno tramite registri	
Entry Point: 0C00H:0006H	
Diagnostica: se il numero di cifre specificate è insufficiente a rappresentare il numero non viene segnalato alcun errore, ma il risultato è inaffidabile	

Conversione da binario a decimale con uscita sul video (cifre decimali codificate ASCII)

Questa procedura dopo aver espletato il servizio della procedura precedente, invia al video le codifiche ASCII delle cifre stesse: le codifiche delle cifre 0 non significative vengono trasformate in codifiche del carattere Blank prima di essere inviate al video

Parametri passati alla procedura	**Registri**
Numero binario da convertire	AX
Indirizzo logico della prima locazione del buffer	DX, BX
Numero delle cifre decimali del risultato (al più 5)	CX
Parametri ritornati al programma chiamante	
Nessuno tramite registri	
Entry Point 0C00H:0009H	
Diagnostica: se il numero di cifre specificate è insufficiente a rappresentare il numero non viene segnalato alcun errore, ma il risultato è inaffidabile.	



Il prezzo della presente pubblicazione è stato determinato calcolando i soli costi di produzione nell'ambito della politica regionale per il diritto allo studio universitario.

L. 3.000 IVA compresa